SQL

Module – 3

Introduction

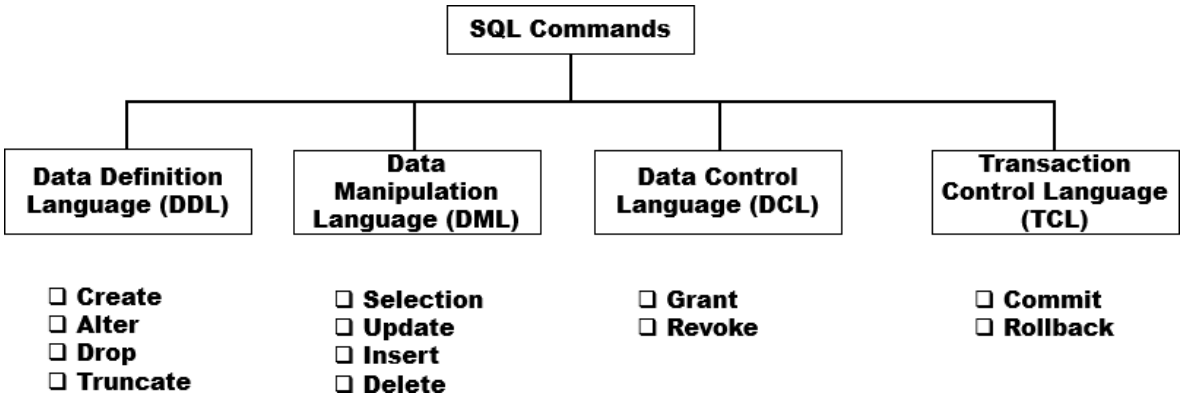
SQL is a Structured Query Language and it also called as SEQUEL (Structured English Query Language) and was designed and implemented at IBM Research. The SQL language may be considered one of the major reasons for the commercial success of relational databases. SQL is a comprehensive database language. It has statements for data definitions, queries, and updates. Hence, it is both a DDL 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. It also has rules for embedding SQL statements into a general-purpose programming language such as Java, COBOL, or C/C++.

Some of the DBMS and RDBMS tools are, MySql, SQLserver, Oracle, Sybase, MS Access, Informix, Postgres etc.

Query means commands (i.e., with these commands we access and manipulate data through table in database).

Note:** Usually we call it as CRUD operation→ Create Read Update Dele
i.e., Create Database, Create Table, Alter Database, Alter Table, Reading Data, Update Data, Delete Database, Delete Table and Delete data all this operations is done using SQL commands.

Categories of SQL Commands



SQL Data Definition and Data Types

SQL uses the terms table, row, and column for the formal relational model terms relation, tuple, and attribute, respectively. The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), domains, views, assertions and triggers.

1. Schema and Catalog concepts in SQL

Creating schema is the first process before learning SQL programing constructs. 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. A schema is created via the **CREATE SCHEMA statement.**

For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier 'Sunil'.

CREATE SCHEMA COMPANY **AUTHORIZATION** 'Sunil';

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.

Catalog → It is one which stores the description of the whole database i.e., description means how many tables they have, attributes do each table have, what are the constraints that are applied on the tables and what are domains of attributes all these constitutes to form a Catalog.

2. The CREATE TABLE Command in SQL

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command.

Typically, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing,

CREATE TABLE COMPANY.EMPLOYEE

Rather than,

CREATE TABLE EMPLOYEE

The relations declared through CREATE TABLE statements are called **base tables.**

Examples:

```
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 DEPT_LOCATIONS
        ( Dnumber           INT                 NOT NULL,
          Dlocation         VARCHAR(15)         NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE PROJECT
        ( Pname             VARCHAR(15)         NOT NULL,
          Pnumber           INT                 NOT NULL,
          Plocation         VARCHAR(15),
          Dnum              INT                 NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );

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

3. Attribute Data Types and Domains in SQL

Basic data types

- **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 - precision, total number of decimal digits
    - j - scale, number of digits after the decimal point
- **Character-string data types**
  - 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.
- **Bit-string data types**
  - **Fixed length n - BIT(n)** or varying length - **BIT VARYING(n)**, where n is the maximum number of bits.
  - **BINARY LARGE OBJECT or BLOB** is also available to specify columns that have large binary values, such as images.
  - The maximum length of a BLOB can be specified in kilobits (K), megabits (M), or gigabits (G)
  - For example, BLOB(30G) specifies a maximum length of 30 gigabits.
- **A Boolean data type**
  - It has the traditional values of TRUE or FALSE. 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.
- **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.
  Only valid dates and times should be allowed by the SQL implementation.
- **TIME WITH TIME ZONE data type** includes an additional six positions for specifying the displacement from the standard universal time zone, which is in the range +13:00 to –12:59 in units of HOURS:MINUTES. If WITH TIME ZONE is not included, the default is the local time zone for the SQL session.

Additional data types

- **Timestamp data type** (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME

ZONE qualifier.

- **INTERVAL data type.** This specifies an interval—a relative value that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

It is possible to specify the data type of each attribute directly or a domain can be declared, and the domain name used with the attribute Specification. 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) for the attributes Ssn and Super_ssn of EMPLOYEE, Mgr_ssn of DEPARTMENT, Essn of WORKS_ON, and Essn of DEPENDENT.

Specifying Constraints in SQL

Basic constraints that can be specified in SQL as part of table creation:

1. **Key constraint**: A primary key value cannot be duplicated
   Dname VARCHAR(15) UNIQUE;

2. **Entity Integrity constraint**: A primary key value cannot be null
   Dnumber INT PRIMARY KEY

A constraint to restrict attribute or domain values using the **CHECK** clause
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 to the following:
Dnumber INT NOT NULL CHECK (Dnumber > 0 AND Dnumber < 21);

3. **Referential integrity constraints** : The "foreign key " must have a value that is already present as a primary key, or may be null.

   FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
   FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber)

   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.
   FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber) **ON DELETE SET DEFAULT ON UPDATE CASCADE**
   FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn) **ON DELETE SET NULL ON UPDATE CASCADE**
   FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber**) ON DELETE CASCADE ON UPDATE CASCADE**

Basic Retrieval Queries in SQL

SQL has one basic statement for retrieving information from a database: the **SELECT statement.**

1. The SELECT-FROM-WHERE Structure of Basic SQL Queries

The basic form of the SELECT statement, sometimes called a mapping or a select-from-where block, 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) expression that identifies the tuples to be retrieved by the query.

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	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	1968-07-19	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	MGRSSN	MGRSTARTDATE
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

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

Example1: Retrieve the name and address of all employees who work for the Research department.

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

(DNAME='Research') is a *selection condition* (corresponds to a SELECT operation in relational algebra)
(DNUMBER=DNO) is a *join condition* (corresponds to a JOIN operation in relational algebra)

Example2: **Retrieve the birthdate and address of the employee whose name is 'John B. Smith**

```
SELECT BDATE, ADDRESS
FROM  EMPLOYEE
WHERE FNAME='John' AND MINIT='B'AND LNAME='Smith'
```

Example3: For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

```
SELECT  PNUMBER, DNUM, LNAME, BDATE, ADDRESS
        FROM        PROJECT, DEPARTMENT, EMPLOYEE
        WHERE DNUM=DNUMBER AND MGRSSN=SSN AND PLOCATION='Stafford'
```

there are *two* join conditions
The join condition DNUM=DNUMBER relates a project to its controlling department
The join condition MGRSSN=SSN relates the controlling department to the employee who manages
that department

---

2. Ambiguous Attribute Names, Aliasing, Renaming, and Tuple Variables

In SQL, the ***same name can be used for two or more attributes*** as long as the attributes are in ***different relations***.

A query that refers to two or more attributes with the same name, 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***.

Example: For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

```
SELECT E.Fname, E.Lname, S.Fname, S.Lname
FROM EMPLOYEE AS E, EMPLOYEE AS S
WHERE E.Super_ssn=S.Ssn;
```

– We can think of E and S as two *different copies* of EMPLOYEE; E represents employees in role of *supervisees* and S represents employees in role of *supervisors*

---

3. Unspecified WHERE Clause and Use of the Asterisk

A missing WHERE clause indicates ***no condition on tuple*** selection; hence, ***all tuples of the relation are selected***. If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—all possible tuple combinations—of these relations is selected.

<u>Query:</u> Retrieve the SSN values for all employees.

- **Q:**
```
SELECT SSN
FROM  EMPLOYEE
```

If more than one relation is specified in the FROM-clause *and* there is no join condition, then the *CARTESIAN PRODUCT* of tuples is selected

USE OF *

To ***retrieve all the attribute values*** of the selected tuples, ***a * is used***, which stands for *all the attributes*

Examples:

Q: SELECT *
FROM EMPLOYEE
WHERE DNO=5

Q: SELECT *
FROM EMPLOYEE, DEPARTMENT
WHERE NAME='Research' AND DNO=DNUMBER

USE OF DISTINCT

- SQL does not treat a relation as a set; *duplicate tuples can appear*
- To eliminate duplicate tuples in a query result, the keyword **DISTINCT** is used
- For example, the result of Q1 may have duplicate SALARY values whereas Q1A does not have any duplicate values

Q1: SELECT SALARY FROM EMPLOYEE
Q1A: SELECT DISTINCT SALARY FROM EMPLOYEE

(a)

Salary
30000
40000
25000
43000
38000
25000
25000
55000

(b)

Salary
30000
40000
25000
43000
38000
55000

INSERT, DELETE and UPDATE Statements in SQL

The INSERT Command

- 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.

Example: *One way to write is,*

INSERT INTO EMPLOYEE VALUES (1, ‘sunil’, ‘kumar’, 100);

Another way to write is,

INSERT INTO EMPLOYEE (USN, Fname, Lname, Dno) VALUES (1, ‘sunil’, ‘kumar’, 100);

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command.

The DELETE Command

- Removes tuples from a relation
- Includes a WHERE-clause to select the tuples to be deleted
- Tuples are deleted from only *one table* at a time (unless CASCADE is specified on a referential integrity constraint)
- A missing WHERE-clause specifies that *all tuples* in the relation are to be deleted; the table then becomes an empty table
- The number of tuples deleted depends on the number of tuples in the relation that satisfy the WHERE-clause
- Referential integrity should be enforced

Example: DELETE FROM EMPLOYEE WHERE Lname= kumar;

Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table.

The UPDATE Command

- Used to modify attribute values of one or more selected tuples
- A WHERE-clause selects the tuples to be modified
- An additional SET-clause specifies the attributes to be modified and their new values
- Each command modifies tuples *in the same relation*
- Referential integrity should be enforced

Example: Change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively.

UPDATE PROJECT **SET** Plocation = _Bellaire‘, Dnum = 5 **WHERE** Pnumber=10;

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

SQL has directly incorporated some of the **set operations** from mathematical set theory, which are also part of relational algebra. There are

- set union (UNION)
- set difference (EXCEPT) and
- set intersection (INTERSECT)

The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result. These set operations apply only to union-compatible relations, so 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.

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

(SELECT DISTINCT Pnumber FROM PROJECT, DEPARTMENT,

```
EMPLOYEE WHERE Dnum=Dnumber AND Mgr_ssn=Ssn AND Lname= kumar)
UNION
SELECT DISTINCT Pnumber FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE Pnumber=Pno AND Essn=Ssn AND Lname= kumar);
```

Substring Pattern Matching and Arithmetic Operators

The first feature allows comparison conditions on only parts of a character string, using the LIKE comparison operator. This can be used for string pattern matching. Partial strings are specified using two reserved characters:

- % replaces an arbitrary number of zero or more characters.
- _(underscore) replaces a single character.

For example: consider the following query: Retrieve all employees whose address is in Mysore.

```
SELECT Fname, Lname FROM EMPLOYEE WHERE Address LIKE %Mysore%;
```

To retrieve all employees who were born during the 1950s, we can use Query

```
SELECT Fname, Lname FROM EMPLOYEE
WHERE Bdate LIKE '1987 _______ ;
```

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: 1987 _____ \:

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (–), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains. For example, suppose that we want to see the effect of giving all employees who work on the ProductX project a 10 percent raise; we can issue the following query:

```
SELECT E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal
FROM EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
WHERE E.Ssn=W.Essn AND W.Pno=P.Pnumber AND P.Pname= ProductX';
```

Between (operator): It is used

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

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

The condition (Salary BETWEEN 30000 AND 40000) 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 of the attributes that appear in the query result, by using the **ORDER BY clause.**

Example: 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.

```
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 DESC if we want to see the result in a descending order of values. The keyword ASC can be used to specify ascending order explicitly. For example, if we want descending alphabetical order on Dname and ascending order on Lname, Fname, the ORDER BY clause can be written as,

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

SQL- Advances Queries

More Complex SQL Retrieval Queries

Additional features allow users to specify more complex retrievals from database.

Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. NULL is used to represent a missing value, but that it usually has one of three different interpretations—value.

Example:

- **Unknown value**. A person‟s date of birth is not known, so it is represented by NULL in the database.
- **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.
- **Not applicable attribute.** An attribute CollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.

Each individual NULL value is considered to be different from every other NULL value in the various database records. When a NULL is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE. It is therefore necessary to define the results (or truth values) of three- valued logical expressions when the logical connectives AND, OR, and NOT are used.

Truth Tables

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

The rows and columns represent the values of the results of comparison conditions, which would typically appear in the WHERE clause of an SQL query.

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause of the query to TRUE are selected. Tuple combinations that evaluate to FALSE or UNKNOWN are not selected.

SQL allows queries that check whether an attribute value is NULL using the comparison operators.

IS NULL or IS NOT NULL

Example: Retrieve the names of all employees who do not have supervisors.

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

Nested Queries, Tuples, and Set/Multiset Comparisons

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.

Example 1: Write a nested query to find the names of the employees who work in RESEARCH Department.

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPEER_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	1941-06-02	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

SELECT fname, Iname FROM employee
WHERE Dno = (SELECT Dnumber FROM department

WHERE Dname = 'Research');

Example 2: Write a nested query to find the names of the departments which are located in HOUSTON.

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

SELECT dname FROM department
WHERE dnumber IN (SELECT dnumber FROM dept locations
WHERE dlocation = "Houston');

Example 3: Write a query to find the names of the employees who have son

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPEER_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	1941-06-02	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

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

SELECT fname, Lname FROM employee
WHERE SSN IN (SELECT essn FROM dependent WHERE relation 'Son')

Example 4: Display the name of faculty who are teaching students from IS Department.

Student

Name	USN	Age	Department	FID
Arun	4CS23	20	CS	10
Arjun	4IS24	21	IS	11
Abhi	4IS25	22	IS	11

Faculty

FID	Name	Age	Salary
10	Rohan	35	20000
11	Ramu	36	25000
12	Ravi	37	30000

SELECT Name
FROM Faculty
WHERE FID **IN** (SELECT FID FROM Student WHERE Department=IS);

Output:

Outer Query result

Name
Ramu
Ramu

Inner Query result

FID
11
11

Note: IN operator can check with multiple values and **= operator** can check with only one value.

Example 5: Find out the names of faculties who are taking salary more than salary taken by the faculty who is working in IS department.

- Fist find out maximum salary taken from IS department
- Then compare with other department.

Student

Name	USN	Age	Department	FID
Arun	4CS23	20	CS	10
Arjun	4IS24	21	IS	11
Abhi	4IS25	22	IS	11
Kitti	4IS26	23	IS	12

Faculty

FID	Name	Age	Salary
10	Rohan	35	20000
11	Ramu	36	25000
12	Ravi	37	30000

```
SELECT Name
FROM Faculty
WHERE Salary < (SELECT Salary FROM Faculty. F, Student. S
                WHERE F.FID = S.FID AND Department = IS);
```

Output:

Inner Query result

Salary
25000
30000

Outer Query result

Name
Rohan

Note:** Attributes can be of multiple ones SELECT Name FROM Faculty WHERE (A1, A2) IN (SELECT A1, A2, A3);

Nested Queries::Comparison Operators

Other comparison operators can be used to compare a single value v to a set or multiset V. 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. 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.

For 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);
```

Correlated Nested Queries

- Correlated nested queries are used for row-by-row processing.
- Each nested sub-query is executed once for every row of the outer query

- 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.
- A correlated nested query is one way of reading every row in a table and comparing values in each row against related data

Example: Find out the names of employees who have their dependent with the same Fname and same gender of employee.

Employee

Fname	Lname	SSN	Salary	Gender
Arun	Kumar	1	20000	M
Abhi	p	2	25000	M
Sunil	Bhat	3	30000	M
Ajay	Kumar	4	32000	M

Dependent

ESSN	Name	Gender	Age
1	Ramu	M	12
2	Rohan	M	14
4	Ajay	M	18

```
SELECT Fname, Lname
FROM Employee.E
WHERE IN (SELECT ESSN FROM Dependent. D WHERE Fname=Name AND E.Gender
= D.Gender);
```

Output:

Fname	Lname
Ajay	Kumar

The EXISTS, NOT EXISTS and UNIQUE Functions in SQL

The EXISTS function in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples) or not. The result of EXISTS is a Boolean value.

- TRUE if the nested query result contains at least one tuple, or
- FALSE if the nested query result contains no tuples.

For example, the query to retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee can be written using EXISTS functions as follows:

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE E
WHERE EXISTS (SELECT * FROM Dependent AS D
WHERE Fname = Name AND E.Gender=D.Gender);
```

Output:

Fname	Lname
Ajay	Kumar

NOT EXISTS Functions

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

Example: Retrieve the names of employees who have no dependents.

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

Output:

Fname	Lname
Sunil	Bhat

Example 2: List the names of managers who have at least one dependent.

Employee

Fname	Lname	SSN	Salary	Gender
Arun	Kumar	1	20000	M
Abhi	p	2	25000	M
Sunil	Bhat	3	30000	M
Ajay	Kumar	4	32000	M

Dependent

ESSN	Name	Gender	Age
1	Ramu	M	12
2	Rohan	M	14
4	Ajay	M	18

Department

Dname	Mgr_SSN	Dlocation
IS	1	Mysore
CS	2	Mysore

```
SELECT Fname, Lname
FROM Employee
WHERE EXISTS (SELECT * FROM Dependent WHERE SSN = Mgr_SSN);
        AND
        EXISTS (SELECT * FROM Dependent WHERE SSN = ESSN);
```

Output:

Fname	Lname
Arun	Kumar
Abhi	p

UNIQUE Functions

UNIQUE returns TRUE if there are no duplicate tuples in the result of query, otherwise, it returns FALSE. This can be used to test whether the result of a nested query is a set or a multiset.

Explicit Sets and Renaming of Attributes in SQL

IN SQL it is possible to use an explicit set of values in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses.

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

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

In SQL, it is possible to rename any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name

Example: Retrieve the last name of each employee and his or her supervisor.

```
SELECT E.Lname AS Employee_name,
 S.Lname AS Supervisor_name
FROM EMPLOYEE AS E, EMPLOYEE AS S
WHERE E.Super_ssn=S.Ssn;
```

Join Tables in SQL and Outer Joins

An SQL join clause combines records from two or more tables in a database. It creates a set that can be saved as a table or used as is. A JOIN is a means for combining fields from two tables by using values common to each. SQL specifies four types of JOIN,

- INNER,
- OUTER
- EQUIJOIN and
- NATURAL JOIN

INNER JOIN

An inner join is the most common join operation used in applications and can be regarded as the default join-type. Inner join creates a new result table by combining column values of two tables (A and B) based upon the join- predicate (the condition). The result of the join can be defined as the outcome of first taking the Cartesian product (or Cross join) of all records in the tables (combining every record in table A with every record in table B)—then return all records which satisfy the join predicate

Example:

```
SELECT * FROM employee
INNER JOIN department ON
employee.dno = department.dnumber;
```

EQUIJOIN and NATURAL JOIN

- An EQUIJOIN is a specific type of comparator-based join that uses only equality comparisons in the join-predicate

NATURAL JOIN is a type of EQUIJOIN where the join predicate arises implicitly by comparing all columns in both tables that have the same column-names in the joined tables. The resulting joined table contains only one column for each pair of equally named columns.

Example: Retrieve the names of every employee who work for IS department.

Employee

SSN	Ename	Salary	dno
1	Rohan	10000	10
2	Ravi	20000	11
3	Ramu	15000	11
4	Sunil	22000	12

Department

DNO	Dname	Mgr_SSN
10	CS	1
11	IS	2
12	CV	4
13	ME	NULL

SSN	Ename	Salary	DNO	Dname	Mgr_SSN
1	Rohan	10000	10	CS	1
2	Ravi	20000	11	IS	2
3	Ramu	15000	11	IS	2
4	Sunil	22000	12	CV	4

```
SELECT Ename
FROM Employee E, Dependent D
WHERE E.dno = D.DNO AND Dname = IS);
```

Output:

Ename
Ravi
Ramu

Note:** Same query can be written for JOIN using FROM Clause.

```
SELECT Ename
FROM Employee JOIN department ON dno = DNO
WHERE Dname = IS;
```

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.

Example: Using natural join we can write query as,

```
SELECT Ename
FROM Employee Natural JOIN department
WHERE Dname = IS;
```

CROSS JOIN returns the Cartesian product of rows from tables in the join. In other words, it will produce rows which combine each row from the first table with each row from the second table.

OUTER JOIN

An outer join does not require each record in the two joined tables to have a matching record. The joined table retains each record-even if no other matching record exists. Outer joins subdivide further into,

- Left outer joins
- Right outer joins
- Full outer joins

No implicit join-notation for outer joins exists in standard SQL.

Left Outer Join

- Every tuple in the left table must appear in the result.
- If no matching in tuple will be padded with NULL values for attributes in the right table.

Example: Display the names of employees who are mangers.

Employee

SSN	Ename	Salary	dno
1	Rohan	10000	10
2	Ravi	20000	11
3	Ramu	15000	11
4	Sunil	22000	12

Department

DNO	Dname	Mgr_SSN
10	CS	1
11	IS	2
12	CV	4
13	ME	NULL

SELECT Ename
FROM Employee Left Outer Join Department ON SSN = Mgr_SSN;

SSN	Ename	Salary	DNO	Dname	Mgr_SSN
1	Rohan	10000	10	CS	1
2	Ravi	20000	11	IS	2
3	Ramu	15000	NULL	NULL	NULL
4	Sunil	22000	12	CV	4

Right Outer Join

- Every tuple in the right table must appear in the result.
- If no matching in tuple will be padded with NULL values for attributes in the left table.

Example: Display the details of employees who are working and not working for in some or different department.

SELECT Ename
FROM Employee AS E Right Outer Join Department AS D ON e.Dno = d. Dno;

Employee

SSN	Ename	Salary	DNO
1	Ravi	10000	10
2	Ramu	22000	11
3	Rohan	10000	11
4	Sunil	20000	12

SSN	Ename	Salary	DNO	Dname	Mgr_SSN
1	Rohan	10000	10	CS	1
2	Ravi	20000	11	IS	2
3	Ramu	15000	11	IS	2
4	Sunil	22000	12	CV	4
NULL	NULL	NULL	13	ME	NULL

Full Outer Join

- A full outer join combines the effect of applying both left and outer joins.
- Where records in full outer joined tables do not match, the result set will have NULL values for every column of the table that lacks a matching row.
- For those records that do not match, a single row will be produced in the result set (containing fields populated from both tables).

Example:

R1

Fname	Lname
A	B
C	D

R2

Fn	Ln
C	D
E	F

Fname	Lname	Fn	Ln
A	B	NULL	NULL
C	D	C	D
NULL	NULL	E	F

SELECT *
FROM R1 Full Outer Join R2 ON Fname = Fn;

Aggregate Functions in SQL

Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary. A number of built-in aggregate 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 can be 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. The functions MAX and MIN can also be used with attributes that have non numeric domains if the domain values have a total ordering among one another.

Example: 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), AVG (Salary)
FROM Employee;

Output:

Min (Salary)
10000
10000

Max (Salary)
22000

Sum (Salary)
62000

Avg (Salary)
15500

SELECT COUNT (SSN) FROM Employee;

Output:

COUNT (SSN)
3

SELECT COUNT (*) FROM Employee;

Output:

COUNT (*)
4

To retrieve the names of all employees who have two or more dependents.

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE (SELECT COUNT (*)
FROM DEPENDENT
WHERE Ssn=Essn) >= 2;
```

Employee

SSN	Ename	Salary	DNO
1	Ravi	10000	10
2	Ramu	22000	11
3	Rohan	10000	11
4	Sunil	20000	12

Dependent

ESSN	DNmae	.....
1	X	
2	Y	
2	Z	
2	P	
3	Q	

Output:

Ename
Ramu

Grouping:

- In many cases, we want to apply the aggregate functions *to subgroups of tuples in a relation*
- Each subgroup of tuples consists of the set of tuples that **have *the same value*** for the

grouping attribute(s)

- The function is applied to each subgroup independently
- It is used in conjunction with **aggregate functions to produce summary reports from the database**
- SQL has a **GROUP BY**-clause for specifying the grouping attributes, which *must also appear in the SELECT-clause*

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

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

Employee

SSN	Ename	Salary	DNO
1	Ravi	10000	10
2	Ramu	22000	11
3	Rohan	10000	11
4	Sunil	20000	12
5	Smith	18000	12

Output:

DNO	COUNT(*)	Avg(Salary)
10	1	10000
11	2	16000
12	2	19000

- **HAVING**

Having is used to restrict the rows affected by the GROUP BY clause. It is similar to the WHERE clause.
The HAVING-clause is used for specifying **a selection condition on groups** (rather than on individual tuples)

Example: Retrieve only those departments and its average salary or more than one employee working.

SELECT Dno, COUNT (*), AVG (Salary)
FROM EMPLOYEE
GROUP BY Dno;
HAVING COUNT (*)>1;

Output:

DNO	COUNT(*)	Avg(Salary)
11	2	16000
12	2	19000

Note:** For using having clause = group by clause is a must.

Specifying Constraints as Assertions and Actions as Triggers

Assertions are used to specify additional types of constraints outside scope of built-in relational model constraints. In SQL, users can specify general constraints via declarative assertions, using the CREATE ASSERTION statement of the DDL. Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.

General form:

CREATE ASSERTION <Name_of_assertion> CHECK (<cond>)

For the assertion to be satisfied, the condition specified after CHECK clause must return true.

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));

Employee E

SSN	Ename	Dno	Salary
1	Ravi	10	20k
2	Ramu	10	30k

Employee M

SSN	Ename	Dno	Salary
1	Ravi	10	20k
2	Ramu	10	30k

Department

Dno	Dname	Mgr_SSN
10	Sunil	2

Output: Condition of assertion was evaluated and no such row found that where in which the salary of the employee is greater than the salary of manager and vice versa.

Not Exists (False)
TRUE

Triggers in SQL

A trigger is a procedure that runs automatically when a certain event occurs in the DBMS. In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. The CREATE TRIGGER statement is used to implement such actions in SQL.

General form:

CREATE TRIGGER <name>
BEFORE | AFTER | <events>
FOR EACH ROW |FOR EACH STATEMENT
WHEN (<condition>)
<action>

A trigger has three components

1. **Event:** When this event happens, the trigger is activated
   - Three event types: Insert, Update, Delete
   - Two triggering times: Before the event
     After the event
2. **Condition (optional):** If the condition is true, the trigger executes, otherwise skipped.
3. **Action:** The actions performed by the trigger.

Example:

Employee

SSN	Ename	Salary	Super_SSN
1	Ravi	20k	2
2	Ramu	21k	3
3	Rohan	22k	2

Constraint is insert a new row to employee table.

4	Sunil	25k	3

Before inserting to this row into table need to verify less or greater and only if less then only I should trigger. Two keyword is used one is NEW and other is OLD.

CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF SALARY, Super_SSN
ON EMPLOYEE
FOR EACH ROW
WHEN (NEW.SALARY > (SELECT SALARY FROM EMPLOYEE
WHERE SSN = NEW. Super_SSN))

INFORM_SUPERVISOR (NEW. Super_SSN, NEW.SSN);

Output:

25k > 22k = TRUE

Perform Action = INFORM_SUPERVISOR (NEW. Super_SSN, NEW.SSN);
Pass (3, 4)

Assertions vs. Triggers

- Assertions do not modify the data, they only check certain conditions. Triggers are more powerful because the can check conditions and also modify the data.
- Assertions are not linked to specific tables in the database and not linked to specific events. Triggers are linked to specific tables and specific events.
- All assertions can be implemented as triggers (one or more). Not all triggers can be implemented as assertions.

VIEWS (Virtual Table) in SQL

A view in SQL terminology is a single table that is derived from other tables. Other tables can be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered to be a virtual table, in contrast to base tables, whose tuples are always physically stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view. We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.

Syntax:

CREATE VIEW Name AS
SELECT Columns
FROM Table Name
WHERE Condition

Example: For single table.

CREATE VIEW Detials AS
SELECT Nmae, Address
FROM Students
WHERE id<118;

Example: For multiple tables.

CREATE VIEW Detials AS
SELECT S.Nmae, S.Address, M.Marks
FROM Students .S, Marks.M
WHERE Sid=Mid;

If we do not need a view any more, we can use the **DROP VIEW command** to dispose of it.

For example: DROP VIEW WORKS_ON;

Schema Change Statements in SQL

Schema evolution commands available in SQL can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements. This can be done while the database is operational and does not require recompilation of the database schema.

The DROP Command

- can be used to drop named schema elements, such as tables, domains, or constraints.
- One can also drop a schema
- There are two drop behavior options: CASCADE and RESTRICT
- **CASCADE**: to remove the COMPANY database schema and all its tables, domains, and other elements.

DROP SCHEMA COMPANY CASCADE;

- **RESTRICT**: schema is dropped only if it has no elements in it; otherwise, the DROP command will not be executed.
- To use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself.

If a **relation** within a schema is no longer needed, the relation and its definition can be deleted by using the **DROP TABLE command**

DROP TABLE DEPENDENT CASCADE;

RESTRICT a table is dropped only if it is not referenced in any constraints or views or by any other elements

The DROP TABLE command not only deletes all the records in the table if successful, but also removes the table definition from the catalog

The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command

- adding or dropping a column
- changing a column definition,
- adding or dropping table constraints.

ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN **Job** VARCHAR **(12);**

ALTER TABLE COMPANY.EMPLOYEE **DROP COLUMN** Address **CASCADE;**

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

ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn DROP DEFAULT;

ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn SET DEFAULT 333445555

Alter Table - Alter/Modify Column

To change the data type of a column in a table, use the following syntax:

ALTER TABLE table_name
MODIFY column_name datatype;

For example we can change the data type of the column named "DateOfBirth" from date to year in the "Persons" table using the following SQL statement:

ALTER TABLE Persons
ALTER COLUMN DateOfBirth year;

Notice that the "DateOfBirth" column is now of type year and is going to hold a year in a two- or four-digit format.

DATABASE APPLICATION DEVELOPMENT

Introduction

We often encounter a situations in which we need the greater flexibility of a general-purpose programming language in addition to the data manipulation facilities provided by SQL. For example, we may want to integrate a database applications with GUI or we may want to integrate with other existing applications.

Accessing Databases from applications

SQL commands can be executed from within a program in a host language such as C or Java. A language to which SQL queries are embedded are called Host language.

1. Embedded SQL

The use of SQL commands within a host language is called Embedded SQL. Conceptually, embedding SQL commands in a host language program is straight forward. SQL statements can be used wherever a statement in the host language is allowed. SQL statements must be clearly marked so that a preprocessor can deal with them before invoking the compiler for the host language. Any host language variable used to pass arguments into an SQL command must be declared in SQL.

There are two complications:

- Data types recognized by SQL may not be recognized by the host language and vice versa
    - This mismatch is addressed by casting data values appropriately before passing them to or from SQL commands.
- SQL is set-oriented
    - Addressed using cursors

Declaring Variables and Exceptions

SQL statements can refer to variables defined in the host program. Such host language variables must be prefixed by a colon (:) in SQL statements and be declared between the commands.

EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION

The declarations are similar to C, are separated by semicolons.
For example: we can declare variables c_sname, c_sid, c_rating, and c_age (with the initial c used as a naming convention to emphasize that these are host language variables) as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

The first question that arises is which SQL types correspond to the various C types, since we have just declared a collection of C variables whose values are intended to be read (and possibly set) in an SQL run-time environment when an SQL statement that refers to them is executed. The SQL-92 standard defines such a correspondence between the host language types and SQL types for a number of host languages. In our example, c_sname has the type CHARACTER(20) when referred to in an SQL statement, c_sid has the type INTEGER, crating has the type SMALLINT, and c_age has the type REAL.

Embedding SQL statements

All SQL statements embedded within a host program must be clearly marked with the details dependent on the host language. In C, SQL statements must be prefixed by EXEC SQL. An SQL statement can essentially appear in any place in the host language program where a host language statement can appear.

Syntax:

EXE SQL → followed by SQL statements

Whereas SQL statements can be any SQL query such as DDL, DML and TCL.

Example: The following embedded SQL statement inserts a row, whose column values are based on the values of the host language variables contained in it, into the sailors relation.

EXEC SQL INSERT INTO sailors VALUES (:c_sname, :c_sid, :c_rating,:c_age);

Cursors

- When you write code for a transaction where the result set includes several rows of data, you must declare and use a cursor.
- Cursor bridges the gap between value-oriented host program and set-oriented DBMS.
- A cursor is a mechanism you can use to fetch rows one at a time.
- An entity that maps over result set and establishes a position on a single row within the result set.
- After the cursor is positioned on a row, operations can be performed on that row, or on a block of rows starting at that position.
- The most common operation is to fetch (retrieve) the current row or block of rows.

Working with a cursor includes the following steps

Declaring the cursor for initializing the memory

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM Sailors S
WHERE S.rating>:c_minrating
```

- **Opening the cursor for allocating the memory**

```
OPEN sinfo;
```

- **Fetching the cursor for retrieving the data**

```
FETCH sinfo INTO :csname, :cage;
```

- **Closing the cursor to release the allocated memory**

```
CLOSE sinfo;
```

The general form of a cursor declaration is:

DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR [WITH HOLD]
FOR some query
[ORDER BY order-item-list]
[FOR READ ONLY I FOR UPDATE]

If the keyword **SCROLL** is specified, the cursor is scrollable, which means that variants of the FETCH command can be used to position the cursor in very flexible ways; otherwise, only the basic FETCH command, which retrieves the next row, is allowed

If the keyword **INSENSITIVE** is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows. Otherwise, and by default, other actions of some transaction could modify these rows, creating unpredictable behavior.

A holdable cursor is specified using the **WITH HOLD** clause, and is not closed when the transaction is committed.

Optional **ORDER BY** clause can be used to specify a sort order.

2. Dynamic SQL

Dynamic SQL Allow construction of SQL statements on-the-fly. Consider an application such as a spreadsheet or a graphical front-end that needs to access data from a DBMS. Such an application must accept commands from a user and, based on what the user needs, generate appropriate SQL statements to retrieve the necessary data. In such situations, we may not be able to predict in advance just what SQL statements need to be executed. SQL provides some facilities to deal with such situations; these are referred to as Dynamic SQL.

SQL provides two main commands PREPARE and EXECUTE.

- PREPARE is a command is used to prepare SQL command which is going to compile and run the query.
- EXECUTE is command which execute command prepared by PREPARE command.

Example:

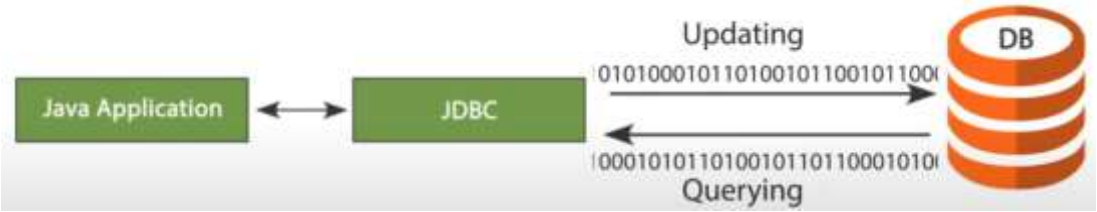
```
char c_sqlstring[] = {"DELETE FROM Sailors WHERE rating>5"};
EXEC SQL PREPARE readytogo
FROM :csqlstring;
EXEC SQL EXECUTE readytogo;
```

- The first statement declares the C variable c_sqlstring and initializes its value to the string representation of an SQL command.
- The second statement results in this string being parsed and compiled as an SQL command, with the resulting executable bound to the SQL variable ready to go.
- The third statement executes the command.

An Introduction to JDBC

ODBC and JDBC, short for Open DataBase Connectivity and Java DataBase Connectivity, also enable the integration of SQL with a general-purpose programming language.

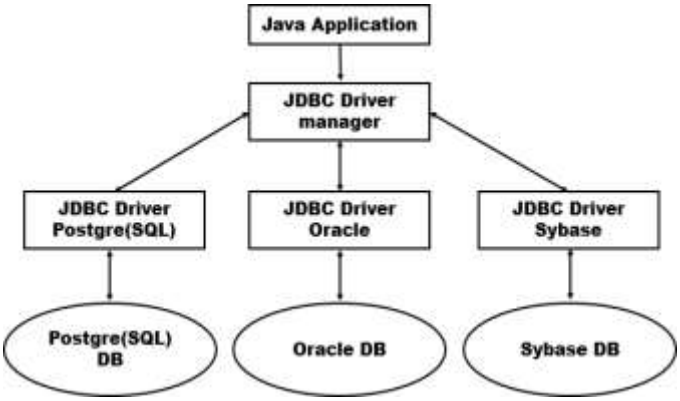
- ODBC and JDBC are DBMS independent.
  - Can access different DBMS simultaneously.
- All direct interaction with a specific DBMS happens through a DBMS-specific driver.
- A driver is a software program that translates the ODBC or JDBC calls into DBMS-specific system calls
- The drivers are managed by driver managers.
- The driver translates the SQL commands from the application into equivalent commands that the DBMS understands.



JDBC Architecture

The architecture of JDBC has four main components:

- Application
- Driver manager
- Drivers
- Data sources



- **Application**
  - Initiates and terminates the connection with a data source.
  - Submits SQL statements and retrieves the results.
- **Driver manager**
  - Load JDBC drivers and pass JDBC function calls from the application to the correct driver.
  - Handles JDBC initialization and information calls from the applications and can log all function calls.
  - Performs some rudimentary error checking.
- **Drivers**
  - Establishes the connection with the data source.
  - Submits requests and returns request results.
  - Translates data, error formats, and error codes from a form that is specific to the data source into the JDBC standard.
- **Data sources**
  - Processes commands from the driver and returns the results.

Types of Drivers in JDBC

Drivers in JDBC are classified into four types depending on the architectural relationship between the application and the data source:

- Type- I driver OR JDBC-ODBC bridge driver.
- Type- II driver OR JDBC-Native-API devices.
- Type- III driver OR Network Protocol driver (JDBC Net Pure Java).
- Type- IV driver OR Thin driver (100% pure java).

Type- I driver OR JDBC-ODBC bridge driver

- The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.

- Advantage:
  - Allows access to almost any database.
- Disadvantage:
  - Not written fully in java - not portable.
  - Poor performance
  - Client system requires ODBC installation to use the driver.

Type- II driver OR JDBC-Native-API devices

- This type of driver translates JDBC function calls directly into method calls of the API of one specific data source.
- The driver is usually, written using a combination of C++ and Java

Advantage

- This architecture performs significantly better than a JDBC-ODBC bridge.

Disadvantage

- The database driver that implements the API needs to be installed on each computer that runs the application.

Type- III driver OR Network Protocol driver (JDBC Net Pure Java)

Network Protocol driver uses a middleware server that translates the JDBC requests into DBMS-specific method calls.

The middleware server can then use a Type II JDBC driver to connect to the data source.

- Advantage:
  - Portable driver
  - Supports caching, load balancing and advanced system administration.
  - Flexible-allows access to multiple databases using one driver.
- Disadvantage:
  - Requires another server application to install and maintain.
  - Traversing the record set may take longer.

Type- IV driver OR Thin driver (100% pure java)

- Instead of calling the DBMS API directly, the driver communicates with the DBMS through Java sockets
- In this case, the driver on the client side is written in Java, but it is DBMS-specific. It translates JDBC calls into the native API of the database system.
- This solution does not require an intermediate layer, and since the implementation is all Java, its performance is usually quite good.

The 7 steps for JDBC are,

- Import the package
  import java.sql.*;
- Load and register the driver
  --class.forname();
- Establish the connection
  Connection con;

- Create a Statement object
  Statement st;
- Execute a query
  st.execute();
- Process the result
- Close the connection.

Step1: Import the package import java.sql.*;
Step2:load the corresponding JDBC driver Class.forName("oracle/jdbc.driver.OracleDriver");
Step 3: create a session with data source through creation of Connection object. Connection connection = DriverManager.getConnection(database_urI, userId, password);
EX: Connection con= DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xesid","system","ambika");

Step 4: create a statement object
JDBC supports three different ways of executing statements:

- Statement
- PreparedStatement and
- CallableStatement.

- The Statement class is the base class for the other two statement classes. It allows us to query the data source with any static or dynamically generated SQL query.
- The PreparedStatement class dynamically generates precompiled SQL statements that can be used several times
- CallableStatement are used to call stored procedures from JDBC. CallableStatement is a subclass of PreparedStatement and provides the same functionality.

Example:
Statement st=con.createStatement();

Step 5: executing a query
String query="select * from students where usn=„4VV15CS001‟";
ResultSet rs=st.executeQuery(query);

Step 6: process the result
String sname=rs.getString(2);
System.out.println(sname);

Step 7: close the connection
con.close();

SQLJ: SQL -JAVA

SQLJ enables applications programmers to embed SQL statements in Java code in a way that is compatible with the Java design philosophy.

Example: SQLJ code fragment that selects records from the Books table that match a given author.

String title; Float price; String author;
#sql iterator Books (String title, Float price);
Books books;
#sql books = {

```
                SELECT title, price INTO:titIe, :price
                FROM Books WHERE author = :author
};
while (books.next()) {
                        System.out.println(books.title() + ", " + books.price());
                    }
books.close() ;
```

All SQLJ statements have the special prefix #sql. In SQLJ, we retrieve the results of SQL queries with iterator objects, which are basically cursors. An iterator is an instance of an iterator class.

Usage of an iterator in SQLJ goes through five steps:

- Declare the Iterator Class: In the preceding code, this happened through the statement
  #sql iterator Books (String title, Float price);
  This statement creates a new Java class that we can use to instantiate objects.
- Instantiate an Iterator Object from the New Iterator Class:
  We instantiated our iterator in the statement Books books;.
- Initialize the Iterator Using a SQL Statement:
  In our example, this happens through the statement #sql books =....
- Iteratively, Read the Rows From the Iterator Object:
  This step is very similar to reading rows through a ResultSet object in JDBC.
- Close the Iterator Object.

There are two types of iterator classes:

- Named iterators
- Positional iterators

For named iterators, we specify both the variable type and the name of each column of the iterator. This allows us to retrieve individual columns by name. This method is used in our above example.

For positional iterators, we need to specify only the variable type for each column of the iterator. To access the individual columns of the iterator, we use a FETCH ... INTO construct, similar to Embedded SQL.

We can make the iterator a positional iterator through the following statement:

#sql iterator Books (String, Float);

We then retrieve the individual rows from the iterator as follows:

```
while (true)
{
#sql { FETCH :books INTO :title, :price,};
if (books.end Fetch())
{ break: }
/ / process the book
}
```

STORED PROCEDURES

Stored procedure is a set of logical group of SQL statements which are grouped to perform a specific task.

Benefits:

- Reduces the amount of information transfer between client and database server.
- Compilation step is required only once when the stored procedure is created. Then after it does not require recompilation before executing unless it is modified and reutilizes the same execution plan whereas the SQL statements need to be compiled every time whenever it is sent for execution even if we send the same SQL statement every time.
- It helps in re usability of the SQL code because it can be used by multiple users and by multiple clients since we need to just call the stored procedure instead of writing the same SQL statement every time. It helps in reducing the development time.

Syntax:

CREATE or Replace Procedure <procedure name> [(arg1 datatype, arg2 datatype)]
Is/As
<Declaration>
Begin

<SQL Statements>
Exception

End procedure name;

Creating a Simple Stored Procedure

Stored procedures enforce strict type conformance: If a parameter is of type INTEGER, it cannot be called with an argument of type VARCHAR. Procedures without parameters are called static procedures and with parameters are called dynamic procedures.

Example: Without using parameters.

CREATE Procedure Get data
SELECT * From Student;

It returns all the attributes in student table.

Example: With using parameters.

CREATE Procedure Get data (IN SID Integer)
SELECT * From Student
WHERE sid = SID;

It returns the columns which is sid =SID.

Stored procedures can also have parameters. These parameters have to be valid SQL types, and have one of three different modes: IN, OUT, or INOUT.

- IN parameters are arguments to the stored procedure.
- OUT parameters are returned from the stored procedure; it assigns values to all OUT parameters that the user can process.
- INOUT parameters combine the properties of IN and OUT parameters: They contain values to be passed to the stored procedures, and the stored procedure can set their values as return values.

Calling Stored Procedures

Stored procedures can be called in interactive SQL with the CALL statement:

CALL stored procedure Name (argl, arg2, .. ,argN);

SQL/PSM (Persistent Stored Modules)

SQL/Persistent Stored Modules is an ISO standard mainly defining an extension of SQL with procedural language for use in stored procedures.

In SQL/PSM, we declare a stored procedure as follows:

CREATE PROCEDURE name (parameter1,... , parameterN)
local variable declarations
procedure code;

We can declare a function similarly as follows:

CREATE FUNCTION name (parameterl, ... , parameterN)
RETURNS sqlDataType
local variable declarations
function code;

Main SQL/PSM Constructs

- We can declare local variables using the DECLARE statement. In our example, we

declare two local variables: 'rating', and 'numOrders'.

- PSM/SQL functions return values via the RETURN statement. In our example, we return the value of the local variable 'rating'.
- We can assign values to variables with the SET statement. In our example, we assigned the return value of a query to the variable 'numOrders'.
- SQL/PSM has branches and loops. Branches have the following form:

```
IF (condition) THEN statements;
ELSEIF statements;
ELSEIF statements;
ELSE statements;
END IF
```

- Loops are of the form.

```
LOOP
    statements:
END LOOP
```

Queries can be used as part of expressions in branches; queries that return a single value can be assigned to variables. We can use the same cursor statements as in Embedded SQL (OPEN, FETCH, CLOSE), but we do not need the EXEC SQL constructs, and variables do not have to be prefixed by a colon **':'**.

Example:

```
CREATE FUNCTION RateCustomer (IN sailorid. INTEGER)
RETURNS INTEGER
DECLARE rating INTEGER;
DECLARE numRes INTEGER;
SET numRes = (SELECT COUNT (*)
                FROM Reserves R
                WHERE R.sid =sailorId);
IF (numRes> 10) THEN rating=1;
ELSEIF rating=O;
END IF;
RETURN rating;
```