Module 3

3.1 MORE COMPLEX SQL QUERIES:

3.1.1Comparisons Involving NULL and Three-Valued Logic:

- **NULL** ***is used to represent a missing value***, ***but that it usually has one of three different interpretations:***

  (i) value unknown (exists but is not known)

  (ii) value not available (exists but is purposely withheld)

  (iii) attribute not applicable (undefined for this tuple).

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

  - **Unknown value*:*** A particular person has a date of birth but it 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 LastCollegeDegree would be NULL for a person who has no college degrees, because it does not apply to that person.

- 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 logic with values ***TRUE or FALSE***.

TABLE 8.1 LOGICAL CONNECTIVES IN THREE-VALUED LOGIC

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

- **Table 8.1** shows the results of three-valued logic.

- ***Rather than using = or <> to compare an attribute value to NULL***, SQL uses **IS** or **IS NOT** key words. **Query 18** illustrates this; its result is shown in **Figure 8.4d**

QUERY 18

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

```
Q18: SELECT FNAME, LNAME
     FROM   EMPLOYEE
     WHERE  SUPERSSN IS NULL;
```

3.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons:

- **Nested Queries** ***are complete select-from-where blocks within the WHERE clause of an outer query***.
- ***SQL has a comparison operator IN, which compares a value 'v' with a set (or multiset) of values 'V' and evaluates to TRUE if 'v' is one of the elements in 'V'.***

```
Q4A: SELECT DISTINCT PNUMBER
     FROM   PROJECT
     WHERE  PNUMBER IN (SELECT PNUMBER
                        FROM   PROJECT, DEPARTMENT,
                               EMPLOYEE
                        WHERE  DNUM=DNUMBER AND
                               MGRSSN=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.
- ✓ 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 a PROJECT tuple if the PNUMBER value of that tuple is in the result of either 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.***

- 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***.

  The following query 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);
```

- To illustrate the potential ambiguity of attribute names in nested queries, consider **Query 16**, whose result is shown in **Figure 8.4c**.

QUERY 16

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

```
Q16:  SELECT  E.FNAME, E.LNAME
      FROM    EMPLOYEE AS E
      WHERE   E.SSN IN  (SELECT  ESSN
                         FROM    DEPENDENT
                         WHERE   E.FNAME=DEPENDENT_NAME
                          AND E.SEX=SEX);
```


3.1.3 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.

```
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;
```

For example,**Q16** may be written as in **Q16A**:

3.1.4 The 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.***

- Query **16B** is an alternative form of query 16 that uses EXISTS.

```
Q16B: SELECT  E.FNAME, E.LNAME
      FROM    EMPLOYEE AS E
      WHERE   EXISTS (SELECT *
                      FROM   DEPENDENT
                      WHERE  E.SSN=ESSN AND E.SEX=SEX
                             AND E.FNAME=DEPENDENT_NAME);
```

We can think of Q16B as follows: For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same social security number, sex, and name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple.

- EXISTS and NOT EXISTS are usually used in conjunction with a correlated nested query.
- In general, ***EXISTS(Q) returns TRUE if there is at least one tuple in the result of the nested query Q, and it returns FALSE otherwise***.
- On theother hand, ***NOT EXISTS(Q) returns TRUE if there are no tuples in the result of nested queryQ, 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);
```

- Following query illustrate the use of NOT EXISTS.

We can explain Q6 as follows: For *each* EMPLOYEE tuple, the correlated nested query selects all DEPENDENT tuples whose ESSN value matches the EMPLOYEE SSN; if the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its FNAME and LNAME.

QUERY 7

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

```
Q7: SELECT FNAME, LNAME
    FROM   EMPLOYEE
    WHERE  EXISTS     (SELECT *
                       FROM   DEPENDENT
                       WHERE  SSN=ESSN)
           AND
           EXISTS     (SELECT *
                       FROM   DEPARTMENT
                       WHERE  SSN=MGRSSN);
```

3.1.5 Explicit Sets and Renaming of Attributes in SQL:

- ***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);
```

- ***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***.
- Hence, the ***AS*** construct can be used to alias both ***attribute*** and ***relation names***, and it can be used in both the ***SELECT*** and ***FROM*** clauses.

- **Q8A** shows how query **Q8** can be slightly changed to retrieve the last name of each employee and his or her supervisor, while renaming the resulting attribute names as EMPLOYEE_NAME and SUPERVISOR_NAME.

  The new names will appear as column headers in the query result.

```
Q8A: SELECT  E.LNAME AS EMPLOYEE_NAME, S.LNAME AS
             SUPERVISOR_NAME
     FROM    EMPLOYEE AS E, EMPLOYEE AS S
     WHERE   E.SUPERSSN=S.SSN;
```


3.1.6 Joined Tables in SQL:

- The concept of a ***joined table (or joined relation)*** was incorporated into SQL ***to permit users to specify a table resulting from a join operation in the FROM clause of a query.***

- For example, consider query *Ql,* which retrieves the name and address of every employee who works for the 'Research' department.

```
Q1A: SELECT  FNAME, LNAME, ADDRESS
     FROM    (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)
     WHERE   DNAME='Research';
```

- The FROM clause in Q1A contains a single *joined table.* The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the

second table, DEPARTMENT.

- 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; an implicit equijoin condition for *each pair of attributes with the same name* from Rand S is created.

- 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;
```

- ***It is also possible to nest join specifications; that is, one of the tables in a join may itself be a joined table.***

  This is illustrated by Q2A, which is a different way of specifying query *Q2,* using the concept of a joined table:

```
Q2A: SELECT  PNUMBER, DNUM, LNAME, ADDRESS, BDATE
     FROM    ((PROJECT JOIN DEPARTMENT ON DNUM=DNUMBER)
             JOIN EMPLOYEE ON MGRSSN=SSN)
     WHERE   PLOCATION='Stafford';
```


3.1.7 Aggregate Functions in SQL:

SQL has a number of built-in aggregate functions: 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***.

- ***The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a total ordering among one another.***

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 following 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.

- In general***, NULL values are discarded when aggregate functions are applied to a particular column (attribute).***

- ***We can specify a correlated nested query with an aggregate function, and then use the nested query in the WHERE clause of an outer query***.

Retrieve the names of all employees who have two or more dependents

```
Q5:  SELECT  LNAME, FNAME
     FROM    EMPLOYEE
     WHERE   (SELECT   COUNT (*)
              FROM     DEPENDENT
              WHERE    SSN=ESSN)  >=  2;
```

3.1.8 Grouping: 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 non overlapping subsets (or groups) of tuples***.

- ***Each group (partition) will consist of the tuples that have the same value of some attributes, called the grouping attributets***.

- 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***.

QUERY 24

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

```
Q24: SELECT   DNO, COUNT (*), AVG (SALARY)
     FROM     EMPLOYEE
     GROUP BY DNO;
```

In Q24, the EMPLOYEE tuples are partitioned into groups-each group having the same value for the grouping attribute DNO. The COUNT and AVG functions are applied to each such group of tuples. **Figure 8.6a** illustrates how grouping works on Q24.It also shows the result of *Q24.*

- ***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

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;
```

- ***Sometimes we want to retrieve the values of these functions only for groups that***

***satisfy certain conditions*.**

For example, suppose that we want to modify **Query 25** so that only projects with more than two employees appear in the result.

- ***SQL provides a HAVING clause, which can appear in conjunction with a GROUP BY clause, for this purpose***.

- ***HAVING provides a condition on the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result ofthe query.***

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;
```

Figure **8.6b** 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;
```

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
               DNO IN    (SELECT    DNO
                          FROM      EMPLOYEE
                          GROUP BY  DNO
                          HAVING    COUNT (*) > 5)
     GROUP BY DNUMBER;
```

- A query in SQL can consist of up to six clauses, but only the first two-SELECT and FROM-are mandatory. The clauses are specified in the following order, with the clauses between square brackets [...] being optional:

```
SELECT <ATTRIBUTE AND FUNCTION LIST>
FROM <TABLE LIST>
[WHERE <CONDITION>]
[GROUP BY <GROUPING ATTRIBUTE(S)>]
[HAVING <GROUP CONDITION>]
[ORDER BY <ATTRIBUTE LIST>];
```

In general, there are numerous ways to specify the same query in SQL. This ***flexibility in specifying queries has advantages and disadvantages***.

- The main ***advantage*** is that users can choose the technique with which they are most comfortable when specifying a query. For example, many queries may be specified with join conditions in the WHERE clause, or by using joined relations in the FROM clause, or with some form of nested queries and the IN comparison operator.

- From the programmer's and the system's point of view regarding query optimization, it is generally preferable to write a query with as little nesting and implied ordering as possible.

- The ***disadvantage*** of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify particular types of queries.

- Another problem is that it may be more efficient to execute a query specified in one way than the same query specified in an alternative way. Thus, an additional burden on the user is to determine which of the alternative specifications is the most efficient.

3.2 SPECIFYING CONSTRAINTS AS ASSERTIONS AND TRIGGERS:

- In SQL, users can specify general constraints - those that do not fall into any of the categories described so far 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.

- 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

```
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.MGRSSN=M.SSN) );
```

for" in SQL, we can write the following assertion:

- 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.
- Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is violated.
- 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.
- Another statement related to CREATE ASSERTION in SQL is CREATE TRIGGER, but triggers are used in a different way.
- Trigger is used to specify the type of action to be taken when certain events occur and when certain conditions are satisfied.
- A typical trigger has three components:
  1. The **event(s)**: These are usually database update operations that are explicitly applied to the database. The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write more than one trigger to cover all possible cases. 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 the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.
  2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event occurs. 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.
  3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.
- Rather than offering users only the option of aborting an operation (using ASSERTION) that causes a violation, the DBMS should make the following option available.
  - It may be useful to specify a condition that, if violated, causes some user to be informed of the violation.

- EX: 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 condition is thus used to monitor the database.
- Other actions may be specified, such as executing a specific stored procedure or triggering other updates.

Trigger to insert a row in DEPT_LOCATIONS when a row is added to DEPARTMENT

```
create trigger t after insert on DEPARTMENT
for each row
        begin
                insert into DEPT_LOCATIONS values (6, 'TEXAS');
        end;
```

```
insert into DEPARTMENT values ('accounts', 6,453453453 '1995-04-23');
```

3.3 VIEWS (VIRTUAL TABLES) IN SQL:

3.3.1 Concept of a View in SQL:

- 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.
- A view does not necessarily exist in physical form; it is considered a virtual table, in contrast to base tables, whose tuples are actually stored in the database.
- For example, we may frequently issue the following query that retrieve the employee name and the project names that the employee works on.

```
SELECT FNAME, PNAME
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE (SSN = ESSN AND PNO =
PNUMBER);
```

  - Rather than having to specify the join of the EMPLOYEE, WORKS_ON, and PROJECT tables every time we issue that query, we can define a view that is a result of these joins.
  - We can then issue queries on the view, which are specified as single-table retrievals rather than as retrievals involving two joins on three tables.
  - We call the EMPLOYEE, WORKS_ON, and PROJECT tables the defining tables of the view.

3.3.2 Specification of Views in SQL:

- 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
  - a query to specify the contents of the view.

- If none of the view attributes results from applying Aggregate functions (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.
- The views in V1 and V2 create virtual tables whose schemas are illustrated in Figure 5.2 when applied to the COMPANY database schema of Figure 5.1.

```
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;
```

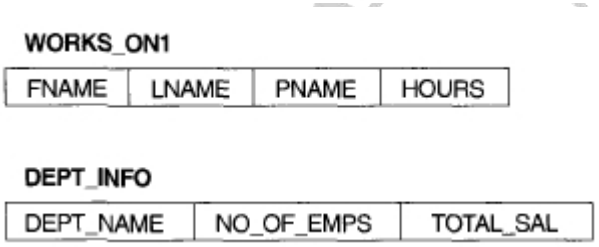

Figure 5.2 Two views V1 and V2 specified on the database schema of Figure 5.1

- We did not specify any new attribute names for the view WORKS_ON1; in this case, WORKS_ON1 inherits the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON.
- We 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.

- We can now specify SQL queries on views V1 and V2 in the same way we specify queries involving base tables.

Retrieve the last name and first name of all employees who work on 'ProjectX'

QV1: SELECT FNAME, LNAME
FROM WORKS_ON1
WHERE PNAME =
'ProjectX' ;

The same query would require the specification of two joins if specified on the base

relations.

- One of the main advantages of a view is to simplify the specification of certain queries. Views are also used as a security and authorization mechanism.
- A view is supposed to be always up to date.
- If we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes.

- It is the responsibility of the DBMS and not the user to make sure that the view is up to date.
- If we do not need a view any more, we can use the **DROP VIEW** command to dispose of it.

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

3.3.3 View Implementation and View Update:

- The problem of efficiently implementing a view for querying is complex. Two main approaches have been suggested for efficient view implementation.
- One strategy, called **query modification**, involves modifying the view query into a query on the underlying base tables.
  For example, the query QV1 would be automatically modified to the following query by the DBMS:

```
SELECT  FNAME, LNAME
FROM    EMPLOYEE, PROJECT, WORKS_ON
WHERE   SSN=ESSN AND PNO=PNUMBER
        AND PNAME='ProjectX';
```

  - The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple queries are applied to the view within a short period of time.
- The other strategy, called **view materialization**, involves physically creating a temporary view table when the view is first queried and keeping that table on the assumption that other queries on the view will follow.
  - In this case, 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 materialized 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.
- Updating of views is complicated and can be ambiguous.
  - An update on a view defined on a single table without any aggregate functions can be mapped to an update on the underlying base table under certain conditions.

- For a view involving joins of multiple base tables, an update operation may be mapped to update operations on the underlying base relations in multiple ways.

➢ To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'. This view update is shown in UVl:

```
UV1: UPDATE  WORKS_ON1
     SET     PNAME = 'ProductY'
     WHERE   LNAME='Smith' AND FNAME='John' AND
             PNAME='ProductX';
```

- This query can be mapped into several updates on the base relations to give the desired update effect on the view. Two possible updates, (a) and (b), on the base relations corresponding to UV1 are shown here:

```
(a): UPDATE  WORKS_ON
     SET     PNO =    (SELECT   PNUMBER
                      FROM     PROJECT
                      WHERE    PNAME='ProductY')
     WHERE   ESSN IN  (SELECT   SSN
                      FROM     EMPLOYEE
                      WHERE    LNAME='Smith' AND FNAME='John')
             AND
             PNO =    (SELECT   PNUMBER
                      FROM     PROJECT
                      WHERE    PNAME='ProductX');

(b): UPDATE  PROJECT SET  PNAME = 'ProductY'
     WHERE   PNAME = 'ProductX';
```


- A view update is feasible when only one possible update on the base relations can accomplish the desired update effect on the view.
- Whenever an update on the view can be mapped to more than one update on the underlying base relations, we must have a certain procedure for choosing the desired update.
- In summary, we can make the following observations:
  - 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.

3.4 EMBEDDED SQL and DYNAMIC SQL:

Impedance Mismatch:

- Impedance mismatch is the term used to refer to the "problems that occur because of differences between the database model and the programming

language model".

- For example, the ***practical relational model has three main constructs***:
  - Attributes and their data types
  - Tuples (rows) and
  - Tables (sets or multisets of records).
- The first problem that may occur is that the "data types of the programming language differ from the attribute data types in the data model".

  Hence, it is necessary to have a binding for each host programming language that specifies for each attribute type the compatible programming language types. It is necessary to have a binding for each programming language because different languages have different data types; for example, the data types available in C and JAVA are different, and both differ from the SQL data types.
- Another problem occurs because the "results of most queries are set (distinct elements) or multisets (duplicated elements) of tuples, and each tuple is formed of a sequence of attribute values.
  - In the program, it is often necessary to access the individual data values within individual tuples for printing or processing.
  - Hence, a binding is needed to map the query result data structure, which is a table, to an appropriate data structure in the programming language.
  - A mechanism is needed to loop over the tuples in a query result in order to access a single tuple at a time and to extract individual values from the tuple.
  - A cursor or iterator variable is used to loop over the tuples in a query result.
  - Individual values within each tuple are typically extracted into distinct program variables of the appropriate type.

3.4.1 Retrieving Single Tuples with Embedded SQL:

- The programming language in which we embed SQL statements language is called the host language such as C, ADA, COBOL, or PASCAL.
- An embedded SQL statement is distinguished from programming language statements by prefixing it with the keywords EXEC SQL so that a preprocessor (or precompiler) can separate embedded SQL statements from the host language code.
- The SQL statements can be terminated by a semicolon (;) or a matching END-EXEC.
- Within an embedded SQL command, we may refer to shared variables which are used in both the C program and the embedded SQL statements.
- Shared variables are prefixed by a colon (:) when they appear in an SQL statement.
- Names of attributes and relations-can only be used within the SQL commands, but shared program variables can be used elsewhere in the C program without the ":" prefix.

- Shared variables are declared within a declare section in the program, as shown in Figure 3.4.1 (lines 1 through 7)

```
0)  int loop ;
1)  EXEC SQL BEGIN DECLARE SECTION ;
2)  varchar dname [16], fname [16], lname [16], address [31] ;
3)  char ssn [10], bdate [11], sex [2], minit [2] ;
4)  float salary, raise ;
5)  int dno, dnumber ;
6)  int SQLCODE ; char SQLSTATE [6] ;
7)  EXEC SQL END DECLARE SECTION ;
```

Figure 3.4.1: c program variables used in the embedded SQL examples E1 and E2

- A few of the common bindings of C types to SQL types are as follows:
  - **i)** The SQL types INTEGER, SMALLINT, REAL, and DOUBLE are mapped to the C types long, short, float,and double, respectively.
  - **ii)** Fixed-length and varying-length strings (CHAR[i], VARCHAR[i]) in SQL can be mapped to arrays of characters (char [i+ 1], varchar [i+ 1]) in C.
- Notice that the only embedded SQL commands in Figure 5.1 are lines 1 and 7, which tell the precompiler to take note of the C variable names between BEGIN DECLARE and END DECLARE. The variables declared in line 6 - SQLCODE and SQLSTATE are used to communicate errors and exception conditions between the database system and the program.

Communicating between the Program and the DBMS Using SQLCODE and SQLSTATE:

- SQLCODE and SQLSTATE are used by the DBMS to communicate exception or error conditions to the application program.
- The SQLCODE variable is an integer variable.
- After each database command is executed, the DBMS returns a value in SQLCODE:
  - **a)** A value of 0 indicates that the statement was executed successfully by the DBMS.
  - **b)** If SQLCODE > 0 (or, more specifically, if SQLCODE = 100), this indicates that no more data (records) are available in a query result.
  - **c)** If SQLCODE < 0, this indicates some error has occurred.
- In later versions of the SQL standard, a communication variable called SQLSTATE was added, which is a string of five characters:
  - **a)** A value of "00000" in SQLSTATE indicates no error or exception.
  - **b)** Other values indicate various errors or exceptions. For example, "02000" indicates "no more data" when using SQLSTATE.

Example of Embedded SQL Programming:

Embedded SQL **loop** that reads a **social security number** of an employee and **prints out** some information from the corresponding **EMPLOYEE record** in the

database.

```
//Program Segment E1:
0)  loop = 1 ;
1)  while (loop) {
2)    prompt("Enter a Social Security Number: ", ssn) ;
3)    EXEC SQL
4)      select FNAME, MINIT, LNAME, ADDRESS, SALARY
5)      into :fname, :minit, :lname, :address, :salary
6)      from EMPLOYEE where SSN = :ssn ;
7)    if (SQLCODE == 0) printf(fname, minit, lname, address, salary)
8)      else printf("Social Security Number does not exist: ", ssn) ;
9)    prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10)   }
```

Figure 3.4.2: Program segment E1, a c program segment with embedded SQL

- The program reads (inputs) a social security number value and then retrieves the EMPLOYEE tuple with that social security number from the database via the embedded SQL command.
- The INTO clause (line 5) specifies the program variables into which attribute values from the database are retrieved.
- C program variables in the INTO clause are prefixed with a colon (:).
- Line 7 in El illustrates the communication between the database and the program through the special variable SQLCODE.
- If the value returned by the DBMS in SQLCODE is 0, the previous statement was executed without errors or exception conditions.
- In El a single tuple is selected by the embedded SQL query; that is why we are able to assign its attribute values directly to C program variables in the INTO clause in line 5.
- In general, an SQL query can retrieve many tuples. In that case, the C program will typically go through the retrieved tuples and process them one at a time. A ***cursor*** is used to allow tuple-at-a-time processing by the host language program. We describe cursors next.

3.4.2 Retrieving Multiple Tuples with Embedded SQL Using Cursors:

- Cursor is a pointer that points to a single tuple (row) from the result of a query that retrieves multiple tuples.
- The cursor is declared when the SQL query command is declared in the program.
- Later in the program, an OPEN CURSOR command fetches the query result from the database and sets the cursor to a position before the first row in the result of the query. This becomes the current row for the cursor. Subsequently, FETCH commands are issued in the program; each FETCH moves the cursor to the next row in the result of the query;
- The cursor variable is basically an iterator that iterates (loops) over the tuples in

the query result-one tuple at a time.

- To determine when all the tuples in the result of the query have been processed, the communication variable SQLCODE (or, alternatively, SQLSTATE) is checked.
- A CLOSE CURSOR command is issued to indicate that we are done with processing the result of the query associated with that cursor.

- Several options can be specified when declaring a cursor. The general form of a cursor declaration is as follows:

```
DECLARE <cursor name> [ INSENSITIVE ] [ SCROLL ] CURSOR
[ WITH HOLD ] FOR <query specification>
[ ORDER BY <ordering specification> ]
[ FOR READ ONLY | FOR UPDATE [ OF <attribute list> ] ] ;
```

Ex:

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

- We can open the cursor using:
  OPEN sinfo:
- We can use the FETCH command to read the first row of cursor sinfo into host language variables:
  FETCH sinfo INTO :csname, :cage;
- By repeatedly executing this FETCH statement (say, in a while-loop in the C program), we can read all the rows computed by the query, one row at a time.

- When we are done with a cursor, we can close it: **CLOSE sinfo;**
- For example, if sinfo is an updatable cursor and open, we can execute the following statement:

```
UPDATE Sailors S
SET S.rating = S.rating 1
WHERE CURRENT of
sinfo;
```


- When the optional keyword SCROLL is specified in a cursor declaration, it is possible to position the cursor in other ways than for purely sequential access.
- A fetch orientation can be added to the FETCH command, whose value can be one of NEXT, PRIOR, FIRST, LAST.
- The fetch orientation allows the programmer to move the cursor around the tuples in the query result with greater flexibility, providing random access by position or access in reverse order.
- If the keyword INSENSITIVE is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows.
- The order-item-list is a list of order-items; an order-item is a column name, optionally followed by one of the keywords ASC or DESC. Every column mentioned in the ORDER BY clause must also appear in the select-list of the

query associated with the cursor.

3.6 Specifying Queries at Runtime Using Dynamic SQL:

- In the previous examples, the embedded SQL queries were written as part of the host program source code. Hence, any time we want to write a different query, we must write a new program, and go through all the steps involved (compiling, debugging, testing, and so on).
- In some cases, it is convenient to write a program that can execute different SQL queries or updates (or other operations) dynamically at runtime.

a) For example, we may want to write a program that accepts an SQL query typed from the monitor, executes it, and displays its result, such as the interactive interfaces available for most relational DBMSs.

b) Another example is when a user-friendly interface generates SQL queries dynamically for the user based on point-and-click operations.

```
//Program Segment E3:
0)   EXEC SQL BEGIN DECLARE SECTION ;
1)   varchar sqlupdatestring [256] ;
2)   EXEC SQL END DECLARE SECTION ;
     …
3)   prompt("Enter the Update Command: ", sqlupdatestring) ;
4)   EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;
5)   EXEC SQL EXECUTE sqlcommand ;
     …
```

- Dynamic SQL example:
  Program segment E3 reads a string that is input by the user (that string should be an SQL update command) into the string variable **sqlupdatestring** in line3. It then prepares this as an SQL command in line 4 by associating it with the SQL variable **sqlcommand.** Line 5 then executes the command.

  It is possible to combine the PREPARE and EXECUTE commands (lines 4 and 5 in E3) into a single statement by writing:

EXEC SQL EXECUTE IMMEDIATE: sqlupdatestring ;

3.7 DATABASE STORED PROCEDURES AND SQL/PSM(Persistent Stored Modules):

- It is sometimes useful to create database program modules (procedures or functions)-that are stored and executed by the DBMS at the database server. These are historically known as database stored procedures, although they can be functions or procedures.
- Stored procedures are useful in the following circumstances:
  a) If a database program is needed by several applications, it can be stored at the server and invoked by any of the application programs. This reduces duplication of effort and improves software modularity.
  b) Executing a program at the server can reduce data transfer and hence communication cost between the client and server in certain situations.
  c) These procedures can enhance the modeling power provided by views by allowing more complex types of derived data to be made available to the database users.

CREATE PROCEDURE ShowNumberOfOrders

SELECT C.cid, C.cname, COUNT(*) FROM Customers C,

Orders a WHERE C.cid = O.cid GROUP BY C.cid, C.cname

Figure 3.7.1 A Stored Procedure in SQL

- 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.

```
CREATE PROCEDURE AddInventory (
                IN book_isbn CHAR(lO),
                IN addedQty INTEGER)
UPDATE Books
        SET     qty_in_stock = qtyjn_stock + addedQty
        WHERE   bookjsbn = isbn
```

Figure 6.9 A Stored Procedure with Arguments

Calling stored procedures in Java

```
CREATE PROCEDURE RallkCustomers(IN number INTEGER)
LANGUAGE Java
EXTERNAL NAME 'file:///c:/storedProcedures/rank.jar'
```

Figure 6.10 A Stored Procedure in Java

```
CALL storedProcedureName(argumentl, argument2, ... , argumentN);
```

Calling stored procedures from JDBC

We can call stored procedures from JDBC using the CallableStatment class. CallableStatement is a subclass of PreparedStatement and provides the same functionality.


```
CallableStatement cstmt=
                con.prepareCall(" {call ShowNumberOfOrders}");
ResultSet rs = cstmt.executeQueryO
while (rs.next())
```

Calling stored procedures from SQLJ

The stored procedure 'ShowNumberOfOrders' is called as follows using SQLJ:

```
// create the cursor class
#sql !terator CustomerInfo(int cid, String cname, int count);

// create the cursor

        CustomerInfo customerinfo;

        // call the stored procedure
        #sql customerinfo = {CALL ShowNumberOfOrders};
        while (customerinfo.nextO) {
            System.out.println(customerinfo.cid() + "," +
                        customerinfo.count());
        }
```

- The general form of declaring a stored procedure in SQL is as follows:

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

We can declare a function similarly as follows:

```
CREATE FUNCTION name (parameter1,..., parameterN)
            RETURNS sqlDataType
    local variable declarations
    function code;
```

The parameters and local declarations are optional, and are specified only if needed.

- If the procedure (or function) is written in a general-purpose programming language, it is typical to specify the language, as well as a file name where the

```
CREATE PROCEDURE <procedure name> (<parameters>)
LANGUAGE <programming language name>
EXTERNAL NAME <file path name> ;
```

program code is stored.

- In general, each parameter should have the following:

a) A parameter mode, which is one of IN, OUT, or INOUT. These correspond to parameters whose values are input only, output (returned) only, or both input and output, respectively.

b) Parameter name.

c) A parameter type that is one of the SQL data types.

- The CALL statement in the SQL standard can be used to invoke a stored procedure

```
CALL <procedure or function name> (<argument list>) ;
```

either from an interactive interface or from embedded SQL or SQLJ. The format of the statement is as follows:

3.7.1 SQL/PSM: Extending SQL for Specifying Persistent Stored Modules:

- **SQL/PSM** is the part of the SQL standard that specifies how to write persistent stored modules.
  - It includes the statements to create functions and procedures that are described in the previous section.
  - It also includes additional programming constructs to enhance the power of SQL for the purpose of writing the code (or body) of stored procedures and functions.
- The **conditional branching statement** in SQL/PSM has the following form:

```
IF <condition> THEN <statement list>
        ELSEIF <condition> THEN <statement list>
        ...
        ELSEIF <condition> THEN <statement list>
        ELSE <statement list>
        END IF ;
```

- Consider the example in **Figure 3.8**, which illustrates how the conditional branch structure can be used in an SQL/PSM function.
  - The function returns a string value (line 1) describing the size of a department based on the number of employees.
  - There is one IN integer parameter, deptno, which gives a department number.
  - A local variable NoOfEmps is declared in line 2.
  - The query in lines 3 and 4 returns the number of employees in the department.
  - The conditional branch in lines 5 to 8 then returns one of the values {"HUGE",

```
    //Function PSM1:
0)  CREATE FUNCTION Dept_size(IN deptno INTEGER)
1)  RETURNS VARCHAR [7]
2)  DECLARE No_of_emps INTEGER ;
3)  SELECT COUNT(*) INTO No_of_emps
4)  FROM EMPLOYEE WHERE Dno = deptno ;
5)  IF No_of_emps > 100 THEN RETURN "HUGE"
6)    ELSEIF No_of_emps > 25 THEN RETURN "LARGE"
7)    ELSEIF No_of_emps > 10 THEN RETURN "MEDIUM"
8)    ELSE RETURN "SMALL"
9)  END IF ;
```

"LARGE", "MEDIUM", "SMALL"} based on the number of employees.

Figure 3.8: Declaring a function in SQL/PSM.

- SQL/PSM has several constructs for looping. There are “while” and “repeat” looping structures, which have the following forms:

```
WHILE <condition> DO
      <statement list>
END WHILE ;
REPEAT
      <statement list>
UNTIL <condition>
END REPEAT ;
```

3.8 THE THREE-TIER APPLICATION ARCHITECTURE

3.8.1 Single-Tier and Client-Server Architectures

Initially, data-intensive applications were combined into a single tier, including the DBMS, application logic, and user interface, as illustrated in Figure 7.5. The application typically ran on a mainframe, and users accessed it through *dumb terminals* that could perform only data input and display. This approach has the benefit of being easily maintained by a central administrator. The commoditization of the PC and the availability of cheap client computers led to the development of the two-tier architecture.

Two-tier architectures, often also referred to as client-server architectures, consist of a client computer and a server computer, which interact through a well-defined protocol. In the traditional client server architecture, the client implements just the graphical user interface, and the server. implements both the business logic and the data management; such clients are often called thin clients, and this architecture is illustrated in Figure 7.6.

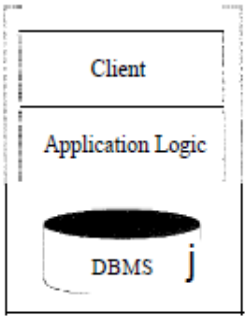

Figure 7.5 A Single-Tier Architecture

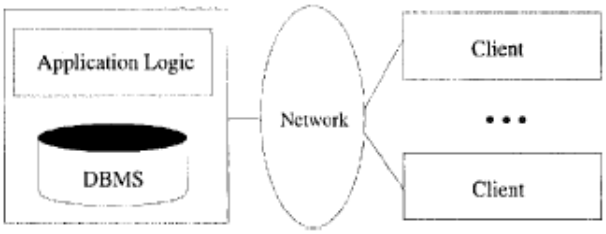

Figure 7.6 A Two-Server Architecture: Thin Clients

3.8.2 THREE TIER ARCHITECTURES

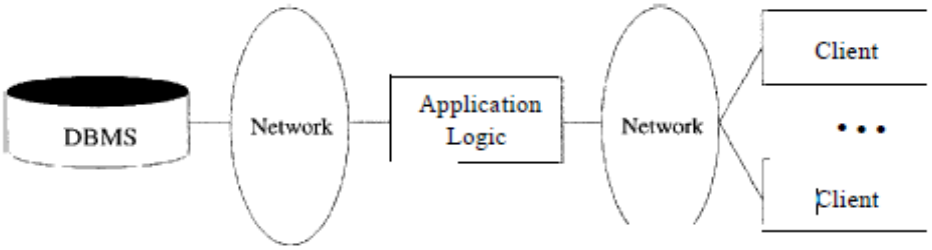

Figure 7.8 A Standard Three-Tier Architecture

Presentation Tier:

- Users require a natural interface to make requests, provide input, and to see results. The widespread use of the Internet has made Web-based interfaces increasingly popular.
- At the presentation layer, we need to provide forms through which the user can issue requests, and display responses that the middle tier generates. Ex: Using The hypertext markup language (HTML)

Middle Tier:

- The application logic executes here. An enterprise-class application reflects complex business processes, and is coded in a general purpose language such as *C++* or Java.
- It controls what data needs to be input before an action can be executed, determines the control flow between multi-action steps, controls access to the database layer, and often assembles dynamically generated HTML pages from

database query results.

Data Management Tier:

- Data-intensive Web applications involve DBMSs, which are the subject of this book.

3.8.3 Advantages of the Three-Tier Architecture

The ***three-tier architecture*** has the following ***advantages***:

a) **Heterogeneous Systems:**
Applications can utilize the strengths of different platforms and different software components at the different tiers. It is easy to modify or replace the code at any tier without affecting the other tiers.

b) **Integrated Data Access:**
In many applications, the data must be accessed from several sources. This can be handled transparently at the middle tier, where we can centrally manage connections to all database systems involved.

c) **Scalability to Many Clients:**
Each client is lightweight and all access to the system is through the middle tier. The middle tier can share database connections across clients, and if the middle tier becomes the bottle-neck, we can deploy several servers executing the middle tier code; clients can connect to anyone of these servers, if the logic is designed appropriately.

d) **Software Development Benefits:**
By dividing the application cleanly into parts that address presentation, data access, and business logic, we gain many advantages.
The business logic is centralized, and is therefore easy to maintain, debug, and change. Interaction between tiers occurs through well-defined, standardized APIs. Therefore, each application tier can be built out of reusable components that can be individually developed, debugged, and tested.

Note: Go through JDBC, SQLJ and Presentation layer from text book (DBMS by Raghu Ramakrishna)