CANARA ENGINEERING COLLEGE
Benjanapadavu – 574219

Program: COMPUTER SCIENCE & ENGINEERING Course code:18CS53
Course Name: DATABASE MANAGEMENT SYSTEM

MODULE 3

Notes prepared by - (Name & Designation) : **Mr.LOHIT B and Mr.SANTOSH**

OBJECTIVE: Design and build database applications for real world problems

OUTCOME: Develop application to interact with databases using JDBC and advanced Queries..

Contents include:

No. of Weblinks: 3
No. of University Qs and As: 10

Structure of Notes

3.1 MORE COMPLEX SQL QUERIES **(VTU Question)**

3.2 SPECIFYING CONSTRAINTS AS ASSERTIONS AND ACTIONS AS TRIGGERS **(VTU Question)**

3.3 VIEWS (VIRTUAL TABLES) IN SQL**(VTU Question)**

3.4 SCHEMA CHANGE STATEMENTS IN SQL**(VTU Question)**

3.5 ACCESSING DATABASES FROM APPLICATIONS

3.6 AN INTRODUCTION TO JDBC. **(VTU Question)**

3.7 JDBC CLASSES AND INTERFACES. **(VTU Question)**

3.8 SQLJ

3.9 STORED PROCEDURE**(VTU Question)**

3.10 SQL/PSM

3.11 THE THREE-TIER APPLICATION ARCHITECTURE**(VTU Question)**

3.12 THE PRESENTATION LAYER. **(VTU Question)**

3.13 HE MIDDLE TIER . **(VTU Question)**

3.14 References

MODULE - 3

3.1 MORE COMPLEX SQL QUERIES

1. Comparisons Involving NULL and Three-Valued Logic:

- NULL is used to represent a missing value that usually has one of the 3 different interpretations.
    - *value unknown* (exists but is not known or it is not known whether a value exists or not),
    - *value not available* (exists but is purposely withheld), or
    - *attribute not applicable* (undefined for this tuple).

Examples:

1. **Unknown value:** A particular person has a date of birth but it is not known, so it is represented by NULL in the database.

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

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

- SQL does not distinguish between the different meanings of NULL.
- In general, each NULL value is considered to be different from every other NULL value in the database records.
- When a record with 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 3-valued logic with values TRUE, FALSE and UNKNOWN.
- The results or truth values of three-valued logical expressions when the logical connectives AND, OR and NOT are used are showed in the table below.

(a)

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

(b)

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

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

In **(a)** and **(b),** the rows and columns represent the values of the results of 2 three valued Boolean expressions which would appear in the *WHERE* clause of an SQL query. Each expression result would have a value of true, false, or unknown.

- In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression of the query to TRUE are selected. Tuple combinations that evaluate to FALSE or UNKNOWN are not selected. There are exceptions to that rule for certain operations such as outer joins.
- SQL allows queries that check whether an attribute value is **NULL.**
- SQL uses the comparison operators **IS** or **IS NOT** to compare an attribute value to NULL. SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate. It follows that when a join condition is specified, tuples with NULL values for the join attributes are not included in the result (unless it is an outer join).

Query 1: ***Retrieve the names of all employees who do not have supervisors.***

```
SELECT Fname, Lname FROM EMPLOYEE
WHERE Super ssn IS NULL;
```

2. 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.
- **Nested queries** are complete select-from-where blocks within another SQL query. That other query is called the **outer query. The nested** queries can **also** appear in the WHERE clause of the FROM clause or other SQL clauses as needed.
- The comparison operator IN compares a value **v with** a set (or multiset) of values **V** and evaluates to TRUE if **v** is one of the elements in V.

Query 4A: Same as query 4.

```
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');
```

- If a nested query returns a single attribute and a single tuple, the query result will be a single value. In such cases, = can be used instead of IN for the comparison operator. In general, a nested query will return a ***table*** (relation) which is a set or multiset of tuples.
- SQL allows the use of tuples of values in comparisons by placing them in parentheses. This is illustrated in the query below.

Query 3:Retrieve the Ssns of all employee who work on the same (project, hours) combination on some project that employee 'John Smith' whose Ssn is 123456789 works on.

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

In this example, the IN operator compares the sub-tuple of values in parentheses (Pno, Hours) for each tuple in WORKS_ON with the set of union compatible tuples produced by the nested query.

- A number of comparison operators can be used to compare a single value v (typically an attribute name) to a set or multiset **V** (typically a nested query). The =ANY (or =SOME) operator returns TRUE if the value v is equal to some value in the set :**V** and is hence equivalent to IN The keywords ANY and SOME have same meaning. Operators that can be combined with ANY include >, >= <, <— and < >. The keyword ALL can also be combined with each of these operators. The comparison condition (v >ALL V) returns TRUE if value 'v is greater than all the values in the set (or multiset) V.

Query 4: Retrieve 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);
```

- If attributes of the same name exist, one in the FROM clause of the nested query and one in the FROM clause of the outer query, then there arises ambiguity in attribute names. *The rule is that the reference to an unqualified attribute refers to the relation declared in the **innermost nested query.*** It is generally advisable to create tuple variables (aliases) for all the tables referenced in an SQL query to avoid errors and ambiguities.

Query 5: Retrieve the name of each employee who has a dependent with the same first name and sex as the employee.

```
SELECT  E.Lname, E.Fname
FROM  EMPLOYEE AS E
WHERE  E.Ssn  IN  (SELECT  Essn
                   FROM  DEPENDENT
                   WHERE  E.Fname = Dependent name AND E.Sex= Sex);
```

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.**
- For a correlated nested query, the *nested query is evaluated once for each tuple (or combination of tuples) in the outer query.*

Example: In Query 5, for each EMPLOYEE tuple, evaluate the nested query, which retrieves the Essn values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the SSN value of the EMPLOYEE tuple is in the result of the nested query, then select that EMPLOYEE tuple.

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. The query below is another way of solving Query ***5.***

```
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;
```

4. The EXISTS and UNIQUE Functions in SQL:

- EXISTS and UNIQUE are Boolean functions that return TRUE or FALSE.
    - They can be used in WHERE clause condition.
    - 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 atleast one tuple or FALSE if the nested query result contains no tuples.
- EXISTS and NOT EXISTS are typically used in conjunction with a correlated nested query.
- EXISTS (Q) returns **TRUE** if there is atleast one tuple in the result of the nested query Q and returns **FALSE** otherwise.
- NOT EXISTS (Q) returns **TRUE** if there are <u>no tuples</u> in the result of the nested query Q and returns **FALSE** otherwise.

Query 5 can be written in an alternative form that uses EXISTS. The nested query references the Ssn, Fname, and Sex attributes of the EMPLOYEE relation from the outer query. For each employee tuple, evaluate the nested query which retrieves all DEPENDENT tuples with the same Essn, Sex and Dependent_name as the employee tuple; if atleast 1 tuple EXISTS in the result of the nested query, then select that employee tuple.

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

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

```
SELECT FNAME, LNAME FROM EMPLOYEE
WHERE NOT EXISTS (SELECT *FROM DEPENDENT
                      WHERE SSN=ESSN);
```

In this query, the correlated nested query retrieves all DEPENDENT tuples related to an EMPLOYEE tuple. If none exist, the EMPLOYEE tuple is selected because the WHERE clause condition will evaluate to TRUE in this case. 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 and so we select that employee tuple.

Query 7: List the names of managers who have atleast one dependent.

```
SELECT  FNAME, LNAME
FROM   EMPLOYEE
WHEREEXISTS (SELECT * FROM      DEPENDENT
                    WHERE SSN=ESSN)
          AND EXISTS (SELECT FROM    DEPARTMENT
                       WHERE   Ssn=Mgrssn);
```

Query 8: Retrieve the name of each employee who works on all the projects controlled by department number 5.

```
SELECT Fname, Lname
FROMEMPLOYEE
WHERENOT EXISTS ((SELECT Pnumber FROM PROJECT WHERE
Dnum=S)
   EXCEPT (SELECT Pno FROM WORKS  ON WHERE Ssn=Essn)) ;
```

*The first subquery (which is not correlated with the outer query) selects all projects controlled bydepartment 5 and the second subquery (which is correlated with the outer query) selects all projects that the particular employee being considered works on. If the set difference of the first subquery result **MINUS EXCEPT)** the second subquery result is empty, it means that the employee works on all the projects and is therefore selected.*

- The function **UNIQUE (Q)** returns TRUE if there are no duplicate tuples in the result of query Q; otherwise it returns FALSE.
- The UNIQUE function can be used to test whether the result of a nested query is a set — no duplicates or a multiset — duplicates exist.

5. Explicit sets and Renaming of Attributes in SQL:

- An explicit set of values can be used in the where clause rather than a nested query. Such a set is enclosed in parentheses in SQL.

Query 8 : Retrieve the Essn 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 **AS** qualifier followed by the desired new name.
- **AS** construct can be used to alias both attribute and relation names in general and it can be used in appropriate parts of a query.

```
SELECT E.Lname AS Employee name, S.Lname AS
Supervisor name FROM EMPLOYEE AS E, EMPLOYEE AS
S
WHERE E.Superssn=S.Ssn;
```

6. Joined Tables in SQL:

- The **joined table** (or **joined relation)** permits users to specify a table resulting from a join operation in *the FROM clause* of a query. This construct avoids mixing together all the select and join conditions in the WHERE clause.

Example: Consider query which retrieves the name and address of every employee who works for the 'Research' department. First specify the join of the EMPLOYEE and DEPARTMENT relations and then select the desired tuples and attributes. The FROM clause contains a single joined table.

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

The attributes of such a table are all the attributes of the first table EMPLOYEE followed by all attributes of the second table DEPARTMENT.

- In a **NATURAL JOIN** on two **relations R** and S, no join condition is specified; an implicit EQUIJOIN condition for each pair of attributes with the same name from R and S is created. Each such pair of attributes is included only once in the resulting relation.

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

Example: Here the DEPARTMENT relation is renamed as DEPT and its attributes are renamed as Dname , Dno, Mssn and Msdate. The implied join condition for this natural join is EMPLOYEE. Dno = DEPT. Dno because it is the only pair of attributes with the same name.

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

- The default type of join in a joined table is an INNER JOIN where a tuple is included in the result only if a matching tuple exists in the other relation. If every tuple should be included in the result, OUTER JOIN must be explicitly specified.

Query 9: Retrieve the names of employees along with their supervisor name and even if employee has no supervisor include his/her name too.

```
SELECT E.Lname AS Employee name, S.Lname AS Supervisor
name FROM (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S ON
E.Superssn=S.Ssn);
```

- **The options available for specifying joined tables in SQL include —**

INNER JOIN – only pairs of tuples that match the join condition are retrieved, same as JOIN.

LEFT OUTER JOIN – every tuple in the left table must appear in the result. If it does not have a matching tuple, it is padded with NULL values for the attributes of the right table.

RIGHT OUTER JOIN . every tuple in the right table must appear in the result. If it does not have a matching tuple, it is padded with NULL values for the attributes of the left table.

FULL OUTER JOIN. The keyword OUTER may be omitted in LEFT OUTER JOIN or

RIGHT OUTER JOIN or FULL OUTER JOIN.

- : • If the join attributes have the same name, we can specify the natural join variation of outer joins by using the keyword **NATURAL** before the operation. Eg: NATURAL LEFT OUTER JOIN.

- : * The keyword CROSS JO IN is used to specify CARTESIAN PRODUCT operation.

- Join specifications can be nested where one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table which is called a **multiway join.**

Example:

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

- Some SQL implementations have a different syntax to specify outer joins by using the comparison operators += for left outer join, =+ for right outer join and +=+ for full outer join when specifying the join condition.(Eg: Oracle uses this syntax)

Example:

```
SELECT E.Lname, S.Lname FROM
EMPLOYEE E, EMPLOYEE S WHERE
E.Super ssn += S.Ssn;
```

7. Aggregate Functions in SQL:

- **Aggregate functions** are used to summarize information from multiple tuples into a single-tuple summary.
- **Grouping** is used to create subgroups of tuples before summarization.
- SQL has 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 the sum, the maximum value, the minimum value and the average of those values respectively.
- 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.
    - NULL values are discarded when aggregate functions are applied to a particular column (attribute). COUNT (*) counts tuples not values hence NULL values do not affect it.
- When an aggregate function is applied to a collection of values, NULLs are removed from the collection before the calculation. If the collection becomes empty because all values are NULL,

the aggregate function will return NULL except COUNT which returns a 0 for an empty collection of values.

- Aggregate functions can also be used in selection conditions involving nested queries. A correlated nested query with an aggregate function can be specified and then used in the WHERE clause of an outer query.
- SQL also has aggregate functions SOME and ALL that can be applied to a collection of Boolean values. SOME returns TRUE if atleast one element in the collection is TRUE whereas ALL returns TRUE if all elements in the collection are TRUE.

Query 10: Find the sum of salaries, maximum salary, the minimum salary, and the average salary among all employees.

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

- This query returns a single-row summary of all the rows in the EMPLOYEE table. We can use the keyword AS to rename the column names in the resulting single-row table.

```
SELECT SUM(SALARY) AS Total_salary, MAX(SALARY) AS
    Highest _salary, MIN(SALARY) AS Lowest salary,
    AVG(SALARY)AS Average salary FROM EMPLOYEE;
```

Query11: Find the sum of the salaries, maximum salary, the minimum salary, and the average salary among employees who work for the 'Research' department.

```
SELECT SUM (SALARY), MAX(SALARY),MIN(SALARY),AVG(SALARY)
FROM EMPLOYEE, DEPARTMENT
WHERE DNO=DNUMBER AND DNAME='Research';
```

Query12: Retrieve the total number of employees in the company.

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

Query13: Retrieve the total number of employees in the 'Research' department.

```
SELECT COUNT (*) FROM  EMPLOYEE, DEPARTMENT
WHEREDNO=DNUMBER AND DNAME='Research';
```

*Here the asterisk (*) refers to the rows (ttiples), so COUNT (*) returns the number of rows in the result of the query. **The** COUNT function can also be used to count values in a column rather than tuples.*

Query14 : Count the number of distinct salary values in the database.

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

COUNT (Salary) will not eliminate duplicate values of Salary. Any tuples with NULL for Salary will not be counted.

Query 15: 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;
```

The correlated nested query counts the number of dependents that each employee has. If the count is greater than or equal to two, the employee tuple is selected.

8. Grouping: The GROUP BY and HAVING Clauses:

- The aggregate functions can be applied *to subgroups of tuples in a relation* where the subgroups are based on some attribute values.

  For e.g., to find the average salary of employees in each department, we need to partition the relation into non-overlapping subsets (or groups) of tuples.

- Each group (or partition) will consist of tuples that have *the same value* of some attribute(s) called the ***grouping attribute(s).The*** function is then applied to each subgroup independently to produce summary information about each group.
- SQL has a GROUP BY-clause for specifying the grouping attributes. These attributes must also appear in the SELECT- clause so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).
- If NULLs exist in the grouping attribute, then a separate group is created for all tuples with a NULL value in the grouping attribute. ***Eg: If the EMPLOYEE tuple had NULL for the grouping attribute Dno, there would be a separate group for those tuples in the result of Query 16.***
- A join condition can be used in conjunction with grouping.
- To retrieve the values of aggregate functions for only those groups that satis.b, certain conditions, SQL provides a HAVING clause which can appear in conjunction with a GROUP BY clause. The HAVING clause is used for specifying a selection condition on groups (rather than on individual tuples) of tuples associated with each value of the grouping attributes. HAVING provides a condition on the summary information regarding the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query.

- • The selection conditions in the WHERE clause limit the tuples to which functions are applied but the HAVING clause serves to choose whole groups.

Query 16: 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;
```

The EMPLOYEE tuples are divided 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 separately. The SELECT clause includes only the grouping attribute and the functions to be applied on each group of tuples.

Query 17: For each project, retrieve the project number, project name, and the number of employees who work on that project.

```
SELECT Pnumber, Pname, COUNT (*FROM PROJECT, WORKS ON
WHERE PNUMBER=PNO GROUP BY Pnumber, Pname;
```

In this case, the grouping and functions are applied after the joining of the two relations.

Query 18 : For each project on which more than two employees work, retrieve the project number, project name, and the number of employees who work on that project.

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

Query 19: For each project, retrieve the project number, the project name and the number of employees from department 5 who work on the project.

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

Query 20: For each department that has more than 5 employees, retrieve the department number and the number of employees who are making a salary more than $40,000.

```
SELECT Dno, COUNT(*) FROM EMPLOYEE
```

```
WHERE Salary>40000 AND Dno IN(SELECT Dno
FROM EMPLOYEE GROUP BY Dno HAVING COUNT(*)>5)
GROUP BY Dno;
```

9. SQL Constructs: WITH and CASE

- **The WITH** clause allows a user to define a table that will only be used in a particular query. This table will be dropped after its use in that query.
- Queries using WITH can generally be written using other SQL constructs.

Example:

```
WITH LARGE_DEPTS (Dno) AS (SELECT Dno FROM EMPLOYEE
GROUP BY Dno HAVING COUNT(*) > 5)
SELECT     Dno, COUNT(*)
FROM       EMPLOYEE
WHERE      Salary>40000 AND Dno IN LARGE DEPTS
GROUP BY Dno;
```

*Here a temporary table LARGE_DEPTS is defined using the **WITH** clause whose result holds the Dnos of departments with more than 5 employees. This table is then used in the subsequent query. Once this query is executed the temporary table LARGE_DEPTS is discarded.*

- The SQL **CASE** construct can be used when a value can be different based on certain conditions.
- It can be used in any part of an SQL query where a value is expected, including when querying, inserting or updating tuples.

Example: Suppose we want to give employees different raise amounts depending on which department they work for. Employees in department 5 get a $2000 raise, those in department 4 get $1500 and those in department 1 get $3000. We can write the update operation as:

```
UPDATE EMPLOYEE
SET Salary = CASE WHEN Dno = 5 THEN Salary + 2000
WHEN Dno = 4 THEN Salary + 1500 WHEN Dno = 1 THEN Salary
+ 3000 ELSE Salary+0;
```

Here the salary raise value is determined through the CASE construct based on the department number for which each employee works.

- The CASE construct can also be used when inserting tuples that can have different attributes being NULL depending on the type of record being inserted into a table, as when a specialization is mapped into a single table or when a union type is mapped into a relation.

10. Recursive Queries in SQL

- A recursive query can be written in SQL using **WITH RECURSIVE** construct. It allows users the capability to specify a recursive query in a declarative manner.
- A recursive relationship between tuples of the same type is the recursive relationship between an employee and supervisor. This relationship is described by the foreign key S up r_s sn of the EMPLOYEE relation.

An example of a recursive operation is to retrieve all supervisees of a supervisor employee e at all levels — all employees e' directly supervised by e, all employees e' directly supervised by each employee e' , all employees e" ' directly supervised by each employee e' and so on.

```
WITH RECURSIVE SUP_EMP (Supssn, Empssn) AS
            ( SELECT Super sn, Ssn FROM EMPLOYEE
            UNION SELECT E. Ssn, S.Supssn

            FROM EMPLOYEE AS E, SUP EMP AS S
            WHEREE.Super ssn=S.Empssn) SELECT *
FROM SUP_EMP;
```

Here the view SUP_EMP will hold the result of the recursive query. The view is initially empty. It is first loaded with the first level (Supervisor, supervisee) Ssn combinations through the first part **(SELECT** Super ssn, Ssn **FROM** EMPLOYEE) which is called the **base query.** This will be combined via UNION with each successive level of supervisees through the second part, where the view contents are joined again with the base values to get the second level combinations which are UNIONed with the first level. This is repeated with successive levels until a fixed pint is reached where no more tuples are added to the view. At this point the result of the recursive query is in the view SUP EMP.

Select SQL Statement:

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:

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

+ The clauses between square brackets are optional.

- The select clause lists the attributes or functions to be retrieved.
- The FROM clause specifies all the relations or tables needed in the query including joined relations

  but not those in nested queries.
- The WHERE clause specifies the conditions for selection of tuples from these relations including join conditions if needed.
- GROUP BY specifies grouping attributes whereas HAVING specifies a condition on the groups being selected rather than on the individual tuples.
- The built in aggregate functions COUNT, SUM, AVG, MIN and MAX are used in conjunction with grouping but they can also be applied to all the selected tuples in a query without the group by clause.
- :* ORDER BY specifies an order for displaying the result of a query. It is applied at the end to sort the query result.
- A query is evaluated conceptually by first applying the FROM clause followed by the WHERE clause and then by the GROUP BY and HAVING.

3.2 SPECIFYING CONSTRAINTS AS ASSERTIONS AND ACTIONS AS TRIGGERS

The CREATE ASSERTION can be used to specify additional types of constraints that are outside the scope of the built-in relational model constraints (primary and unique keys, entity integrity and referential integrity). These built-in constraints can be specified in CREATE TABLE statement of SQL. The CREATE TRIGGER can be used to specify automatic actions that the database systems will perform when certain events and conditions occur. This type of functionality is referred to as **active databases.**

1. Specifying General Constraints as Assertions in SQL:

- 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.

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

*The constraint name SALARY_CONSTRAINT is followed by the keyword **CHECK** which is followed by a condition in parentheses that must hold true on every database state for the assertion to be satisfied*.

- The constraint name can be used later to refer to the constraint or modify or drop it.
- The DBMS is responsible for ensuring that the condition is not violated.

- Any WHERE clause condition can be used but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions.
- Whenever some tuples in the database cause the condition in the ASSERTION to evaluate to FALSE, the constraint is violated. The constraint is satisfied by a database state if no combination of tuples in that database state violates the constraint.
- To write an assertion, 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 so that the condition will always be TRUE. Thus the assertion is violated if the result of the query isn't empty.
- The CHECK clauses on attributes, domains and tuples are, checked in SQL only when tuples are inserted or updated in a specific table. Hence constraint checking can be implemented more efficiently by DBMS in these cases. The schema designer should use CHECK on attributes, domains and tuples only when sure that the constraint can only be violated by insertion or updating of tuples and use CREATE ASSERTION only in cases where it is not possible to use CHECK on attributes, domains or tuples so that checks are implemented efficiently by DBMS.

2. **Trigger** in **SQL:**

- The **CREATE TRIGGER** statement is used to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. For e.g., it may be useful to specify a condition that, if violated, causes some user to be informed of the violation. 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.

Example: Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database. Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure SALARYLVIOLAT ION, which will notify the supervisor. The trigger could then be written as below.

```
CREATE TRIGGER SALARYLVIOLATION
BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR SSN ON
EMPLOYEE FOR EACH ROW
WHEN (NEW. SALARY > (SELECT SALARY FROM EMPLOYEE
                     WHERE SSN = NEW.SUPERVISOR SSN))
     INFORM SUPERVISOR(NEW.Supervisor ssn, NEW.Ssn );
```

The trigger is given the name `SALARY VIOLATION,` which can be used to remove or deactivate the trigger later.

- **A typical trigger which is regarded as an ECA (Event, Condition, Action) rule has three components:**

  - 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.

  1. 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.

  . 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. In this example, the action is to execute the stored procedure INFORM SUPERVISOR.

- Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates, and updating derived data automatically.
- **A trigger** specifies an **event,** a **condition** and an **action. The** action is to be executed automatically if the condition is satisfied when the event occurs.

```
CREATE TRIGGER <trigger name>
 (AFTER/ BEFORE ) <triggering events> ON table
 name [FOR EACH ROW ]
 [WHEN <condition>]
<trigger actions>
```

3.3 VIEWS (VIRTUAL TABLES) IN SQL

1. Concept of a View in SQL:

- A view in SQL is a single table that is derived from other tables which 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 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.
- A view is a way of specifying a table that we need to reference frequently, even though it may not exist physically.
- Queries can be specified on a view which is specified as single table retrievals.

2. Specification of Views in SQL:

- A view is specified by the SQL command **CREATE VIEW.**
- The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view.
- If none of the view attributes results from applying functions or arithmetic operations, attribute names for the view need not be specified, since they would be the same as the names of the attributes of the defining tables in the default case.
- The view WORKS ON VIEW does not have new attribute names as it inherits the names of the view attributes from the defining tables EMPLOYEE, PROJECT and WORKS ON.

```
CREATE VIEW WORKS ON VIEW
AS SELECT FNAME, LNAME, PNAME, HOURS
     FROMEMPLOYEE, PROJECT, WORKS ON
     WHERESSN=ESSN AND PNO=PNUMBER;
```

- The view DEPT INFO explicitly specifies new attribute names 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.

```
CREATE VIEW     DEPT INFO(DEPT NAME, NO OF EMP, TOTAL SAL)
 AS SELECT      DNAME, COUNT(*), SUM(SALARY)
      FROM      DEPARTMENT, EMPLOYEE
      WHERE     DNUMBER=DNO
      GROUP BY  DNAME;
```

❖ Queries can be specified on views just as specifying queries involving base tables.

Example: To retrieve the last name and first name of all employees who work on `**ProductX**' project.

```
QV: SELECT Fname,Lname
       FROM WORKS ONI
       WHERE Pname=' ProductX' ;
```

- ❖ Advantages of view: It simplifies the specification of certain queries. It is also used as a security and authorization mechanism.
- ❖ A view should always be *up to date* i.e., if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes. Hence a view is not realized at the time of *view definition* but when we *specify a query* on the view.
  - + It is the responsibility of the DBMS to ensure that a view is up-to-date and not of the user to ensure that the view is up-to-date.
- ❖ If a view is not needed, it can be removed by **DROP VIEW** command.

Eg: **DROP VIEW** WORKS ON VIEW;

3. View Implementation and View Update:

- Two main approaches have been suggested to know how efficiently DBMS implements a view for efficient querying.
- The strategy of **query modification** involves modifying or transforming the view query into a query on the underlying base tables.

***Example:* The query QV would automatically be modified to the following query by the DBMS.**

```
SELECT FNAME, LNAME
FROM EMPLOUEE, 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 time.

- The other strategy, **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. Here, an efficient strategy to automatically update the view when the base tables are updated must be developed to keep the view up- to- date. ***Incremental update*** has been developed to determine what new tuples must be inserted, deleted or modified in a materialized view table when a change is applied to one of the defining base tables. The view is generally kept as a materialized (physically stored) table 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 table and recomputed from scratch when future queries reference the view.

- Different strategies as to when a materialized view is updated are possible.
    - ***immediate* update** strategy updates a view as soon as the base tables are changed.
    - **lazy update** strategy updates the view when needed by a view query.
    - **periodic update** strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date).

  + A retrieval query against any view can always be issued. But issuing an INSERT, DELETE, or UPDATE command on a view table is in many cases not possible.

- In general, 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, an update operation may be mapped to update operations on the underlying base relations in *multiple ways*. Hence, it is not possible for the DBMS to determine which of the updates is intended.
- ***Example:* Suppose that we issue the command to update the Pname attribute of 'John Smith' from ProductX' to 'Productr in the view WORKS_ONVIEW. This view update is shown in UV1: *UV'* :** **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. Some of these updates will create additional side effects that affect the result of other queries. ***Two possible updates, (a) and (b), on the base relations corresponding to UVI are shown. Update (a) relates 'John Smith' to the 'ProductY' PROJECT tuple in place of the 'ProductX' PROJECT tuple and is the most likely desired update.***

(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';
```

Update (b) would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the PROJECT relation to 'ProductY'. It is quite unlikely that the user who specified the view update UVI wants the update to be interpreted as in (b), since it also has the side effect of changing all the view tuples with Pname 'ProductX'.

- Some view updates may not make much sense. ***For example, modifying the Total_Sal attribute of the DEPT_INFO view does not make sense because Total Sal is defined to be the sum of the individual employee salaries. This request is shown as UV2:***

```
UV2: UPDATE    DEPT_INFO
SET  Total Sal=100000
WHERE Dname= 'Research ;
```

- 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, it is usually not permitted.
- 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.
- In SQL, the clause **WITH CHECK OPTION** should be added at the end of the view definition if a view *is to be updated* by INSERT, DELETE, or UPDATE statements. This allows the system to reject operations that violate the SQL rules for view updates.
- It is also possible to define a view table in the **FROM clause** of an SQL query. This is known as an **in-line view.**

4. Views as Authorization Mechanisms:

- Views can be used to hide certain attributes or tuples from unauthorized users.
- ***Suppose a certain user is only allowed to see employee information for employees who work for department 5;*** then we can create the following view DEPTEMP and grant the user the privilege to query the view but not the base table EMPLOYEE itself. This user will only be able to retrieve employee information for employee tuples whose Dno = 5, and will not be able to see other employee tuples when the view is queried.

```
CREATE VIEW DEPTEMP AS SELECT * FROM EMPLOYEE
WHERE Dno = 5;
```

A view can restrict a user to only see certain columns; ***for example, only the first name, last name, and address of an employee may be visible as follows:***

```
CREATE VIEW BASIC _EMP DATA AS SELECT Fname, Lname, Address FROM
EMPLOYEE;
```

- By creating an appropriate view and granting certain users access to the view and not the base tables, they would be restricted to retrieving only the data specified in the view.

3.4 SCHEMA CHANGE STATEMENTS IN SQL The schema evolution commands available in SQL can be used to alter a schema by adding or dropping tables, attributes, constraints and other schema elements.

1. The DROP Command:

- Used to drop *named* schema elements, such as tables, domains, or constraints.
- A whole schema can be dropped if it is not needed by using the **DROP SCHEMA.** command.
- There are two *drop behavior* options: CASCADE and RESTRICT.

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

```
DROP SCHEMA COMPANY CASCADE;
```

- If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has no elements in it. Else the DROP command will not be executed if the schema has elements.
- ***If a base relation within a schema is not needed any longer, the relation and its definition can be deleted by using the DROP TABLE command. The DEPENDENT relation can be removed by issuing the following command:***

```
DROP TABLE DEPENDENT CASCADE;
```

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

- The DROP command can also be used to drop other types of named schema elements, such as constraints or domains.
- The DROP TABLE command not only deletes all the records in the table if successful, but also removes the table definition from the catalog.

2. The ALTER Command:

- The definition of a base table or of other named schema elements can be changed by using the ALTER command.
- For base tables, the possible alter table actions include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints.

  For example, an attribute for keeping track of jobs of employees to the EMPLOYEE base relation can be added in the COMPANY schema by using the command

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

  A value for the new attribute JOB for each individual EMPLOYEE tuple must be entered. This can be done either by specifying a default clause or by using the UPDATE command individually on each tuple. If no default clause is specified, the new attribute will have NULLS in all the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is *not allowed* in this case.

- A column can be dropped by choosing either the CASCADE or RESTRICT for drop behavior. If cascade is chosen, all constraints and views that reference the column are dropped automatically from the schema along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints reference the column.

  For example, the attribute Address can be removed from the EMPLOYEE base table by using the following command.

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

  A column definition can be altered 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 ‘12345’;**

- A constraint specified on a table can be changed by adding or dropping a constraint. A constraint can be dropped if it had been given a name when it was specified. For example, the constraint named EMPSUPERFK can be dropped from the EMPLOYEE relation by

  ALTER TABLE COMPANY. EMPLOYEE **DROP CONSTRAINT** EMPSUPERFK **CASCADE;**

- We can redefine a replacement constraint by adding a new constraint to the relation, if needed. This is specified by using the ADD keyword in the ALTER TABLE statement followed by the new constraint which can be named or unnamed and can be of any of the table constraint types.

```
ALTER TABLE COMPANY. EMPLOYEE ADD CONSTRAINT EMPSUPERFK FOREIGN KEY(SUPER
SSN) REFERENCES EMPLOYEE(SSN);
```

3.5 ACCESSING DATABASES FROM APPLICATIONS

The use of SQL commands within a host language program is called **Embedded SQL.** Details of Embedded SQL also depend on the host language.

1. Embedded SQL:

- SQL statements (i.e. not declarations) can be used wherever statement in the host language is allowed (with a few restrictions).
- SQL statements must be early marked so that a pre-processor can deal with them before invoking the compiler for the host language.
- Any host language variables used to pass arguments into an SQL command must be declared in SQL. Some special host language variables must be declared in SQL. For example, any error conditions arising during SQL execution can be communicated back to the main application program in the host language.
- The data types recognized by SQL may not be recognized by the host language and vice versa. This mismatch is typically addressed by casting data values appropriately before passing them to or from SQL commands. SQL being set-oriented is 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.
- Host-language variables must be declared between the commands EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION. The declarations are separated by semicolons. For example, we can declare variables c_sname, c_sid, c_rat ing, 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 SQL-92 standard defines a correspondence between the host language types and SQL types for a number of host languages. ***In the example, c sname has the type CHARACTER (20) when referred to in an SQL statement, c_sid has the type INTEGER, c rating has the type SMALLINT, and c_age has the type REAL***.
- The SQL-92 standard recognizes two special variables for reporting errors when executing an SQL statement, i.e. SQLCODE and SQLSTATE.

- SQLCODE is defined to return some negative value when an error condition arises, without specifying further just what error a particular negative integer denotes.
- SQLSTATE associates predefined values with several common error conditions, thereby introducing some uniformity to how errors are reported. One of these two variables must be declared.
- The appropriate C type of SQLCODE is long and the appropriate C type of SQLSTATE is char [6] , that is, a character string five characters long.

Embedding SQL Statements:

- All SQL statements embedded within a host program must be clearly marked. 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.

  ***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);`
- A semicolon terminates the command, as per the convention for terminating statements in C.
- The SQLSTATE variable should be checked for errors and exceptions after each Embedded SQL statement. SQL provides the WHENEVER command to simplify this tedious task.

EXEC SQL WHENEVER [SQLERROR I NOT POUND] [CONTINUE I GOT) stmt]

- The intent is that the value of SQLSTATE should be checked after each Embedded SQL statement is executed. If SQLERROR is specified and the value of SQLSTATE indicates an exception, control is transferred to stmt, which is presumably responsible for error and exception handling. Control is also transferred to stmt if NOT FOUND is specified and the value SQL STATE is 02000, which denotes NO DATA.

2. Cursors:

- The impedance mismatch problem occurs when embedding SQL statements in a host language like C, because SQL operates on set of records, whereas languages like C do not cleanly support a set-ofrecords abstraction. The solution is to provide a mechanism that allows us to retrieve rows one at a time from a relation. This mechanism is called a cursor.
- A cursor can be declared on any relation or on any SQL query.
- Once a cursor is declared we can-
  1) **open** it which positions the cursor just`before the first row
  2) **fetch** the next row
  3) **move** the cursor (to the next row, to the row after next n, to the first row, or to the previous row, etc., by specifying additional parameters for the FETCH command)

4) **close** the cursor.

- A cursor essentially allows us to retrieve the rows in a table by positioning the cursor at a particular row and reading its contents.

Basic Cursor Definition and Usage:

- Cursors enable us to examine, in the host language program, a collection of rows computed by an Embedded SQL statement.
- A cursor needs to be opened if the embedded statement is a SELECT (i.e. a query). Opening a cursor can be avoided if the answer contains a single row.
- INSERT, DELETE and UPDATE statements typically require no cursor although some variants of DELETE and UPDATE use a cursor.

Example:

1. We can find the name and age of a sailor, specified by assigning value to the host variable c s id as follows:

```
EXEC SQL SELECT S.sname, S.age INTO :c_sname,:c_age
             FROM Sailors S
             WHERE S.sid = C_sid
```

The **INTO** clause allows us to assign the columns of a single answer row to the host variables c s name and c age. Therefore, we do not need a cursor to embed this query in a host language program.

2. Consider a query, which computes the names and ages of all sailors with a rating greater than the current value of the host variable c minrat ing.

```
SELECT S . sname, S.age
 FROM Sailors S
WHERE S. rating > :c_minrating;
```

This query returns a collection of rows, not just one row. Hence a cursor needs to be used.

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

This code can be included in a C program, and once it is executed, the cursor **sinfo** is defined. Subsequently, we can open the cursor:

OPEN sinfo;

The value of c minrating in the SQL query associated with the cursor is the value of this variable when we open the cursor. (The cursor declaration is processed at compile-time, and the OPEN command is executed at run-time.)

- ***When a cursor is opened, it is positioned just before the first row. We can use the FETCH command to read the first row of cursor sinfo into host language variables:***

```
FETCH sinfo INTO :csname, :c age;
```

- When the FETCH statement is executed, the cursor is positioned to point at the next row (which is the first row in the table when FETCH is executed for the first time after opening the cursor) and the column values in the row are copied into the corresponding host variables. 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. Additional parameters to the FETCH command allow us to position a cursor in very flexible ways.

- The special variables SQLCODE and SQLSTATE indicate when we have looked at all the rows associated with the cursor.
- When we are done with a cursor, we can close it:

```
CLOSE sinfo;
```

- It can be opened again if needed, and the value of : c_minrating in the SQL query associated with the cursor would be the value of the host variable c_minrating at that time.

Properties of Cursors:

- The general form of cursor declaration is:

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

- A cursor can be declared to be-
  - a **read-only cursor** (FOR READ ONLY) or,
  - an **updatable cursor** (FOR UPDATE) if it is a cursor on a base relation or an updatable view.
- If it is updatable, simple variants of the UPDATE and DELETE commands allow us to update or delete the row on which the cursor is positioned.

  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;
```

This Embedded SQL statement modifies the rating value of the row currently pointed to by cursor sinfo. ***We can delete this row by executing the next statement:***

```
DELETE Sailors S
WHERE CURRENT of sinfo;
```

- A cursor is updatable by default unless it 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.

  For example, while we are fetching rows using the s info cursor, we might modify rating values in Sailor rows by concurrently executing the command.

  ```
  UPDATE Sailors S SET S.rating = S.rating-1;
  ```

  Consider a Sailor row such that

  - It has not yet been fetched, and
  - Its original rating value would have met the condition in the WHERE clause of the query associated with sinfo, but the new rating value does not.

  If INSENSITIVE is specified, the behaviour is as if all answers were computed and stored when sinfo was opened; thus, the update command has no effect on the rows fetched by sinfo if it is executed after s in f o is opened. If INSENSITIVE is not specified, the behaviour is implementation dependent in this situation.

- A holdable cursor is specified using the WITH HOLD clause, and is not closed when the transaction is committed. This is needed for a long transaction in which we access (and possibly change) a large number of rows of a table. We can break the transaction into several smaller transactions. The application program can commit the transaction it initiated while retaining its handle on the active table (i.e. the cursor).
- The order in which FETCH command retrieves rows, in general is unspecified, but the optional ORDER BY clause can be used to specify a sort order. The columns mentioned in'the ORDER BY clause cannot be updated through the cursor.
- 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, otherwise it is not clear what columns we should sort on. The keywords ASC or DESC that follow a column control whether the result should be sorted-with respect to that column-in ascending or descending order; the default is ASC. This clause is applied as the last step in evaluating the query.

3. Dynamic SQL:

- An application such as a spreadsheet or a graphical front-end that needs to access data from a DBMS, accepts commands from a user and, based on the user needs, generates appropriate SQL statements to retrieve the necessary data. In such situations, it is not possible to predict in advance which SQL statements need to be executed. SQL provides Dynamic SQL to deal with such situations.
- **The two main commands, PREPARE and EXECUTE are used:**

```
char c_sqlstring[] = {"DELETE FROM Sailors WHERE rating > 5"};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
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 readytogo. The third statement executes the command.

- The preparation of a Dynamic SQL command occurs at run-time and is run-time overhead.
- Interactive and Embedded SQL commands can be prepared once at compile-time and then re-executed as often as desired.
  - ➢ The use of Dynamic SQL should be limited to situations in which it is essential.

3.6 AN INTRODUCTION TO JDBC

- When SQL is embedded in a general-purpose programming language, a DBMS-specific preprocessor transforms the Embedded SQL statements into function, calls in the host language.
- The source code can be compiled to work with different DBMSs but the final executable works only with one specific DBMS.
- ODBC(Open DataBase Connectivity) and JDBC(Java DataBase Connectivity), also enable the integration of SQL with a general-purpose programming language. They expose database capabilities to the application programmer through an application programming interface (API).
- In contrast to Embedded SQL, ODBC and JDBC allow a single executable to access different DBMSs without recompilation. Thus, while Embedded SQL is DBMS-independent only at the source code level, applications using ODBC or JDBC are DBMS-independent at the source code level and at the level of the executable. Using ODBC or JDBC, an application can access several DBMSs simultaneously.
- ODBC and JDBC achieve portability at the level of the executable by introducing an extra level of indirection.
- 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 calls.
- Drivers are loaded dynamically on demand since the DBMSs the application is going to access are known only at run-time. Available drivers are registered with a driver manager. It is sufficient that the driver translates the SQL commands from the application into equivalent commands that the DBMS understands.

- A data storage subsystem with which a driver interacts is referred to as a data source.
- An application that interacts with a data source through ODBC or JDBC selects a data source, dynamically loads the corresponding driver, and establishes a connection with the data source.
- An application can have several open connections to different data sources. Each connection has transaction semantics - changes from one connection are visible to other connections only after the connection has committed its changes.
- While a connection is open, transactions are executed by submitting SQL statements, retrieving results, processing errors, and finally committing or rolling back.
- The application disconnects from the data source to terminate the interaction.

1. Architecture:

- **The architecture of JDBC has four main components:**
    1. The application - initiates and terminates the connection with a data source. It sets transaction boundaries, submits SQL statements, and retrieves the results-all through a well-defined interface as specified by the JDBC API.
    2. The driver manager - load JDBC drivers and pass JDBC function calls from the application to the correct driver. The driver manager also handles JDBC initialization and information calls from the applications and can log all function calls. In addition, the driver manager performs some rudimentary error checking.
    3. Several data source specific drivers - The driver establishes the connection with the data source. In addition to submitting requests and returning request results, the driver translates data, error formats, and error codes from a form that is specific to the data source into the JDBC standard.
    4. Corresponding data sources - The data source processes commands from the driver and returns the results.
- Depending on the relative location of the data source and the application, several architectural scenarios are possible.
- Drivers in JDBC are classified into four types depending on the architectural relationship between the application and the data source:
    - **Type I-Bridges:** This type of driver translates JDBC function calls into function calls of another API that is not native to the DBMS. An example is the JDBC-ODBC bridge; an application can use JDBC calls to access an ODBC compliant data source. The application loads only one driver, the bridge. Bridges have the advantage that is easy to piggyback the application onto an existing installation, and no new drivers have to be installed. But using bridges has several drawbacks. The increased number of layers between data source and application affects performance. In addition, the user is limited to the functionality that the ODBC driver supports.

- **Type II-Direct Translation to the Native API via Non-Java Driver:** This type of driver translates JDBC function calls directly into method invocations of the API of one specific data source. The driver is usually written using a combination of C++ and Java; it is dynamically linked and specific to the data source. This architecture performs significantly better than a JDBC-ODBC bridge. One disadvantage is that the database driver that implements the API needs to be installed on each computer that runs the application.
- **Type III - Network Bridges:** The driver talks over a network to a middleware server that translates the JDBC requests into DBMS-specific method invocations. In this case, the driver on the client site (i.e., the network bridge) is not DBMS-specific. The JDBC driver loaded by the application can be quite small, as the only functionality it needs to implement is sending of SQL statements to the middleware server. The middleware server can then use a Type II JDBC driver to connect to the data source.
- **Type IV-Direct Translation to the Native API via Java Driver:** 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.

3.7 JDBC CLASSES AND INTERFACES

- JDBC is a collection of Java classes and interfaces that enables database access from programs written in the'Java language.
- It contains methods for connecting to a remote data source, executing SQL statements, examining sets of results from SQL statements, transaction management, and exception handling.
- The classes and interface are part of the **j ava . sql** package.
- The package j avax sql adds the capability of connection pooling and the RowSet interface.

1. JDBC Driver Management:

- In JDBC, data source drivers are managed by the Drivermanager class which maintains a list of all currently loaded drivers.
- The Drivermanager class has methods registerDriver, deregisterDriver, and getDrivers to enable dynamic addition and deletion of drivers.
- The first step in connecting to a data source is to load the corresponding JDBC driver. This is accomplished by using the Java mechanism for dynamically loading classes.
- The static method forName in the Class class returns the Java class as specified in the argument string and executes its static constructor. The static constructor of the dynamically loaded class loads an instance of the Driver class, and this Driver object registers itself with the DriverManager class. ***The following Java example code explicitly loads a JDBC driver:***

```
Class.forName("oracle/jdbc.driver.OracleDriver");
```

2. Connections:

- A session with a data source is started through creation of a Connection object.
- A connection identifies a logical session with a data source; multiple connections within the same Java program can refer to different data sources or the same data source.
- Connections are specified through a JDBC URL, a URL that uses the jdbc protocol. Such a URL has the form

```
jdbc:<subprotocol>:<otherParameters>
```

- In JDBC, connections can have different properties. For example, a connection can specify the granularity of transactions. If auto commit is set for a connection, then each SQL statement is `considered to be its own transaction. If autocommit is off, then a series of statements that compose a transaction can be committed using the commit O method of the Connection class, or aborted using the rollback ( ) method. The Connection class has methods to set the autocommit mode (Connection . setAutoCommit) and to retrieve the current autocommit mode (getAutoCommit).
- ***The example shown in figure establishes a connection to an oracle database assuming that the strings user id and password are set to valid values.***

```
    String url = "jdbc:oracle:www.bookstore.com:3083"
    Connection connection;
             try
             {
    connection =
DriverManager.getConnection(url,userid,password);
             }
             catch(SQLExceptionexcpt)
             {
            System.out.println(excpt.getMessage()
            );
             }
```

- The following methods are part of the Connection interface and permit setting and getting other properties:

- public int getTransactionIsolation ( ) throws SQLException and public void setTransactionIsolation (int level) throws SQLException.

  These two functions get and set the current level of isolation for transactions handled in the current connection. All five SQL levels of isolation are possible, and argument 1 can be set as follows:

- TRANSACTION NONE
- TRANSACTION READ UNCOMMITTED
- TRANSACTION READ COMMITTED
- TRANSACTION REPEATABLE READ
- TRANSACTION SERIALIZABLE

- public boolean getReadOnly () throws SQLException and public void setReadOnly (boolean readOnly) throws SQLException. These two functions allow the user to specify whether the transactions executed through this connection are read only.

- public Boolean isClosed ( ) throws SQLException . Checks whether the current connection has already been closed.

- setAutoCommit(Boolean b) and getAutoCommit().

Establishing a connection to a data source involves several steps, such as establishing a network connection to the data source, authentication, and allocation of resources such as memory. In case an application establishes many different connections from different parties (such as a Web server), connections are often **pooled** to avoid this overhead. A **connection pool** is a set of established connections to a data source. Whenever a new connection is, needed, one **of** the connections from the pool is used, instead of creating a new connection to the data source.

- Connection pooling can be handled either by specialized code in the application or the optional j avax sql package, which provides functionality for connection pooling and allows us to set different parameters, such as the capacity of the pool, and shrinkage and growth rates.

3. Executing SQL Statements:

- In JDBC code examples, we assume that we have a Connection object named con.
- **JDBC** supports three different ways of executing statements:
  - **Statement** The **Statement** class is the base class for PreparedStatement class and CallableStatement class. It allows us to query the data source with any static or dynamically generated SQL query.

  - **PreparedStatement** — The **PreparedStatement** class dynamically generates precompiled SQL statements that can be used several times. These SQL statements can have parameters, but their structure is fixed when the **PreparedStatement** object representing the SQL statement is created.

  - CallableStatement Consider the sample code using a **PreparedStatement** object shown in Figure 6.3. The SQL query specifies the query string with ? ' for the values of the parameters, which are set later using methods se tS t r ing, set Float, and set Int. The '?'

placeholders can be used anywhere in SQL statements where they can be replaced with a value - in the WHERE clause (e.g. 'WHERE author =?'), or in SQL UPDATE and INSERT statements, as **in Figure6.3.**

```
   String sql = "INSERT INTO Books VALUES(?,?,?)";
   PreparedStatement pstmt = con.prepareStatement(sql);
  pstmt.setString(1, isbn);
  pstmt.setString(2, title);
  pstmt.setString(3, author);
   int numRows = pstmt.executeUpdate();
```

- The method setString is one way to set a parameter value; analogous methods are available for int, float, and date. It is good to always use clearParameters ( ) before setting parameter values in order to remove any old data
- Different ways of submitting the query string to the data source.

1. **executeUpdate** method - used if we know that the SQL statement does not return any records (SQL UPDATE, INSERT, ALTER, and DELETE statements). The executeUpdate method returns an integer indicating the number of rows the SQL statement modified. It returns 0 for successful execution without modifying any rows.
2. **executeQuery** method - used if the SQL statement returns data, such as in a regular SELECT query. JDBC has its own cursor mechanism in the form of a ResultS et object.
3. **execute** method - more general than executeQuery and executeUpdate.

4. ResultSets:

- The statement executeQuery returns a ResultSet object, which is similar to a cursor.
- ResultSet cursors in JDBC 2.0 allow forward and reverse scrolling and in-place editing and insertions.
- The ResultSet object allows us to read one row of the output of the query at a time.
- Initially, the Result Set is positioned before the first row, and we have to retrieve the first row with an explicit call to the next () method.
- The next method returns false if there are no more rows in the query answer, and true otherwise.
- The code fragment shown in below illustrates the basic usage of a ResultSet object.

```
ResultSet rs = stmt executeQuery (sqlQuery) ;
 String sqlQuery;
ResultSet rs = stmt.executeQuery(sqlQuery)
while (rs.next())
```

- next ( ) allows us to retrieve the logically next row in the query answer.
- To move about in the query answer in other ways:
  - previous ( ) moves back one row.
  - absolute (int num) moves to the row with the specified number.
  - relative (int num) moves forward or backward (if num is negative) relative to the current position. relative(4) has the same effect as previous.
  - first ( ) moves to the first row, and last ( ) moves to the last row.

Matching Java and SOL Data Types:

- JDBC provides special data types and specifies their relationship to corresponding SQL data types. Figure 6.5 shows the accessor methods in a ResultSet object for the most common SQL datatypes.
- With the accessor methods, we can retrieve values from the current row of the query result referenced by the ResultSet object.
- There are two forms for each accessor method –

1. retrieves values by column index, starting at one

2. retrieves values by column name.

SQL Type	Java Class	ResultSet get method
BIT	Boolean	getBoolean()
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
FLOAT	Double	getDouble()
INTEGER	Integer	getInt()
REAL	Double	getFloat()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.TimeStamp	getTimestamp()

Figure 6.5: Reading SQL Datatypes from a `ResultSet` Object

The following example shows how to access fields of the current ResultSet row using accesssor methods.

```
ResultSet rs=stmt.executeQuery(sqlQuery);
String sqlQuery;
ResultSet rs = stmt.executeQuery(sqlQuery)
while (rs.next())
 {
isbn = rs.getString(1);
title = rs.getString("TITLE");
}
```

5. Exceptions and Warnings:

- The methods in j ava . sql can throw an exception of the type SQLException if an error occurs.
- The information includes SQLState, a string that describes the error (e.g., whether the statement contained an SQL syntax error).
- In addition to the standard getMessage () method inherited from Throwable, SQLException has two additional methods that provide further information, and a method to get (or chain) additional exceptions:
    - **`public String getSQLStat ( )`** returns an `SQLState` identifier based on the SQL:1999 specification.
    - **`public int getErrorCode ( )`** retrieves a vendor-specific error code.
    - `public SQLException getNextException` () gets the next exception in a chain of exceptions associated with the current `SQLException` object.

- An SQLWarning is a subclass of SQLException.
- Warnings are not as severe as errors and the program can usually proceed without special handling of warnings.
- Warnings are not thrown like other exceptions, and they are not caught as part of the try-catch block around a java . sql statement. We need to specifically test whether warnings exist.
- Connection, Statement, and ResultSet objects all have a getWarnings () method with which we can retrieve SQL warnings if they exist.
- Duplicate retrieval of warnings can be avoided through clearWarnings () .
- Statement objects clear warnings automatically on execution of the next statement. ResultSet objects clear warnings every time a new tuple is accessed.
- Typical code for obtaining SQLWarnings looks similar to the code shown below.

```
try {
stmt = con.createStatement();
warning = con.getWarnings();
while( warning != null)
warning = warning. getNextWarning ( )
  con . clearWarnings ( ) ;
stmt . executeUpdate ( queryString ;
warning = stmt . getWarnings ( ) ;
while ( warning ! = null )
warning = warning .getNextWarning () ;
 }catch ( SQLException SQLe)

}
```

3.8 SQLJ

- SQLJ (short for 'SQL-Java') was developed by the SQLJ Group, to complement the dynamic way of creating queries in JDBC with a static model.
- Unlike JDBC, having semi-static SQL queries allows the compiler to perform SQL syntax checks, strong type checks of the compatibility of the host variables with the respective SQL attributes, and consistency of the query with the database schema-tables, attributes, views, and stored procedures-all at compilation time.
- In both SQLJ and Embedded SQL, variables in the host language always are bound statically to the same arguments, whereas in JDBC, we need separate statements to bind each variable to an argument and to retrieve the result.
  - ***Example: SQLJ statement binds host language variables title, price, and author to the return values of the cursor books.***

```
#sql books — {
        SELECT title, price INTO :title, :price
        FROM Books WHERE author = :author };
```

- In JDBC, we can dynamically decide which host language variables will hold the query result.

- ***In the following example, we read the title of the book into variable ft it le if the book was written by Feynman, and into variable ot it le otherwise:***

```
// assume we have a ResultSet cursor rs;
author = rs.getString(3);
if (author=="Feynman")
     ftitle = rs.getString(2);
else
     otitle = rs.getString(2);
```

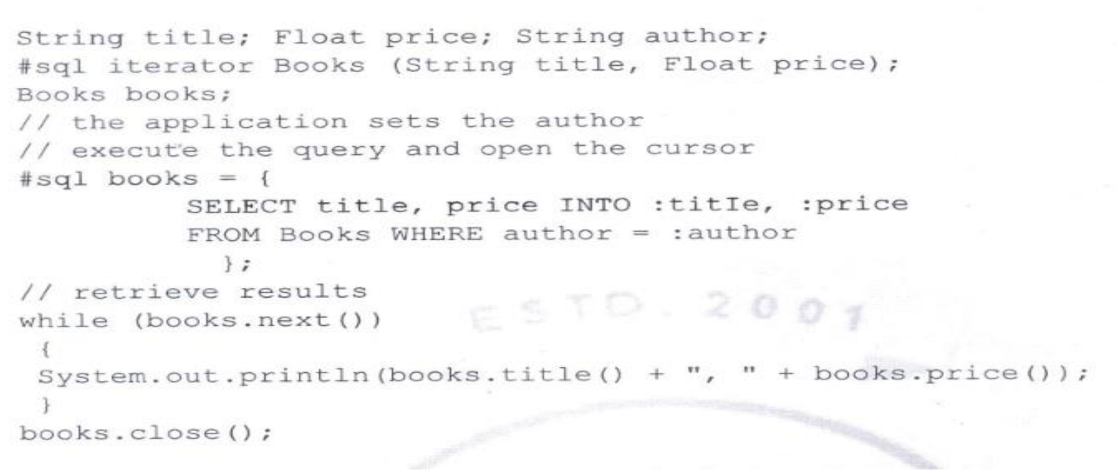
- When writing SQLJ applications, we just write regular Java code and embed SQL statements
- according to a set of rules.

- SQLJ applications are pre-processed through an SQLJ translation program that replaces the embedded SQLJ code with calls to an SQLJ Java library. The modified program code can then be compiled by any Java compiler.
- Usually the SQLJ Java library makes calls to a JDBC driver, which handles the connection to the database system.

1. Writing SQLJ Code:

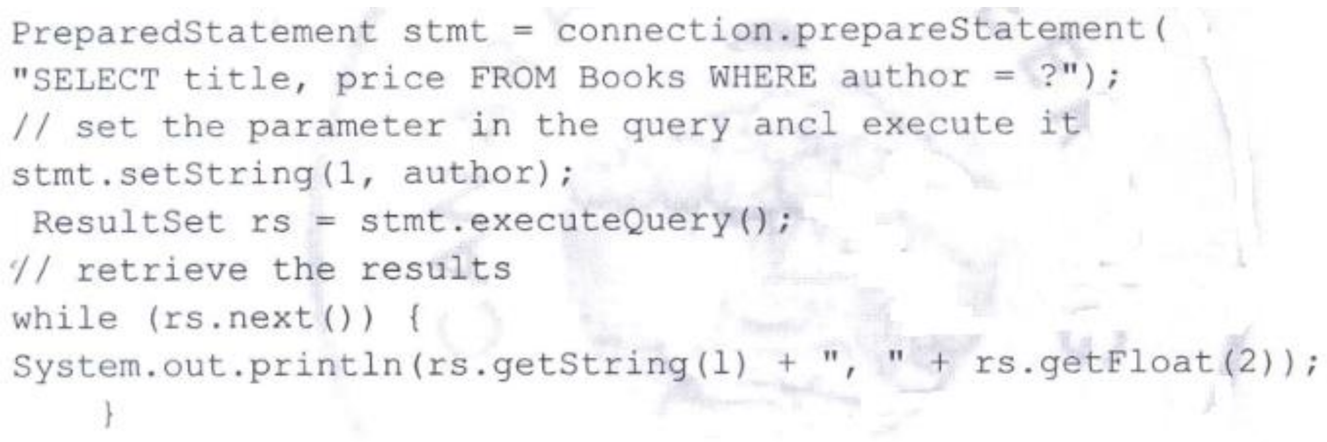
- SQLJ code fragment that selects records from the Books table that match a given author.

The corresponding JDBC code fragment looks as follows (assuming we also declared price, name, and author:



```
String title; Float price; String author;
#sql iterator Books (String title, Float price);
Books books;
// the application sets the author
// execute the query and open the cursor
#sql books = {
            SELECT title, price INTO :title, :price
            FROM Books WHERE author = :author
              };
// retrieve results
while (books.next())
 {
 System.out.println(books.title() + ", " + books.price());
 }
books.close();
```

❖ Comparing the JDBC and SQLJ code, the SQLJ code is easily readable than the JDBC code. Thus, SQLJ reduces software development and maintenance costs.



```
PreparedStatement stmt = connection.prepareStatement(
"SELECT title, price FROM Books WHERE author = ?");
// set the parameter in the query ancl execute it
stmt.setString(1, author);
 ResultSet rs = stmt.executeQuery();
// retrieve the results
while (rs.next()) {
System.out.println(rs.getString(1) + ", " + rs.getFloat(2));
    }
```

- 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:**

    `#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 # *s* ql 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:** 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. In the example, we could retrieve the title column from the Books table using the expression books . title ( ) .
  - **positional iterators:** 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.

Both iterator types have the same performance; which iterator to use depends on the programmer.

- The iterator can be made as 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 . endFetch. ( ) ) {
    break;
     } }
```

3.9 STORED PROCEDURES

- ❖ When SQL statements are issued from a remote application, the records in the result of the query need to be transferred from the database system back to the application. If we use a cursor to remotely access the results of an SQL statement, the DBMS has resources such as locks and memory tied up while the application is processing the records retrieved through the cursor.
- ❖ A **stored procedure** is a program that is executed through a single SQL statement that can be locally executed and completed within the process space of the database server. The results can be packaged into one big result and returned to the application, or the application logic can be performed directly at the server, without having to transmit the results to the client at all.
- ➢ Once a stored procedure is registered with the database server, different users can *reuse the* stored procedure, eliminating duplication of efforts in writing SQL queries or application logic, and making code maintenance easy.
- ❖ Application programmers do not need to know the database schema if we encapsulate all database access into stored procedures.

1. Creating a Simple Stored Procedure:

- ❖ Stored procedures must have a name. Otherwise, it just contains an SQL statement that is precompiled and stored at the server.

```
CREATE PROCEDURE ShowNumberOfOrders
SELECT C.cid, C.cname, COUNT(*)
     FROM Customers C, Orders 0
     WHERE C.cid = O.cid
     GROUP BY C.cid, C.cname
```

Figure: 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** 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 contain values to be passed to the stored procedures, and the stored procedure can set their values as return values.
- Stored procedures enforce strict type conformance: If a parameter is of type **INTEGER,** it cannot be called with an argument of type **VARCHAR.**
- The stored procedure `Addlnventory` has two arguments: `book isbn` and `addedQty.` It updates the available number of copies of a book with the quantity from a new shipment.

```
CREATE PROCEDURE Addlnventory (
         IN book isbn CHAR(10),
         IN addedQty INTEGER)
UPDATE Books
SET    qty in_stock = qty_in_stock + addedQty
WHERE book isbn = isbn
```

Figure: A Stored Procedure with Arguments

- Stored procedures do not have to be written in SQL. They can be written in any host language. As an example, the stored procedure `RankCustomers` is a Java function that is dynamically executed by the database server whenever it is called by the client:

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

2. Calling Stored Procedures:

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

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

In Embedded SQL, the arguments to a stored procedure are usually variables in the host language. For example, the stored procedure **AddInventory** would be called as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char isbn[10];
long qty;
EXEC SQL END DECLARE SECTION
// set isbn and qty to some values
EXEC SQL CALL AddInventory(:isbn,:qty);
```

Calling Stored Procedures from JDBC:

❖ Stored procedures can be called from JDBC using the **CallableStatment** class. `CallableStatement` is a **subclass** of **PreparedStatement.**

❖ A stored procedure could contain multiple SQL statements or a series of SQL statements. Thus, the result could be many different `ResultSet` objects.

❖ The case when the stored procedure result is a single ResultSet is illustrated below.

```
CallableStatement cstmt=con.prepareCall("{call ShowNumber0fOrders}");

  ResultSet rs = cstmt.executeQueryY0 while (rs.next())
```

3. SQL/PSM:

- The SQL/PSM standard is a representative of most vendor specific languages.
- In PSM, we define modules, which are collections of stored procedures, temporary relations, and other declarations
- In SQL/PSM, a stored procedure is declared as follows:

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

- In SQL/PSM, a function is declared as follows:

```
CREATE FUNCTION name (parameterl,..., parameterN)
     RETURNS sq1DataType
 local variable declarations
 function code;
```

+ Each parameter is a triple consisting of the mode (IN, `OUT,` or `INOUT),` the parameter name, and the SQL datatype of the parameter.

Example: A SQL/PSM function that illustrates SQL/PSM constructs.

The function takes as input a customer identified by her cid and a year The function returns the rating of the customer, which is defined as follows:

- Customers who have bought more than ten books during the year are rated 'two';
- customers who have purchased between 5 and 10 books are rated 'one'.
- otherwise the customer is rated 'zero'.

The following SQL/PSM code computes the rating for a given customer and year

```
CREATE PROCEDURE RateCustomer
          (IN custId INTEGER, IN year INTEGER)
           RETURNS INTEGER
DECLARE rating INTEGER;
DECLARE numOrders INTEGER;
SET numOrders =
      (SELECT COUNT(*) FROM Orders 0 WHERE 0.cid = custId);
IF (numOrders>10) THEN rating=2;
```

```
        ELSEIF (numOrders>5) THEN rating=l;
        ELSE rating¯-0;
        END IF;
        RETURN rating;
```

- Some SQL/PSM constructs are:
  - **DECLARE: Local variables can be declared using the DECLARE statement. In the example, we declare two local variables: 'rating', and 'numOrders'.**
  - **RETURN:** PSM/SQL functions return values via the **RETURN** statement. In the example, we return the value of the local variable 'rating'.
  - **SET :** Values can be assigned to variables with the **SET** statement. In our example, we assigned the return value of a query to the variable ᵗnumOrdersᵗ.
  - Branches have the following form:

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

ELSEIF statements;
ELSE statements;
END IF
```

3.11 THE THREE-TIER APPLICATION ARCHITECTURE

Data-intensive Internet applications can be understood in terms of three different functional components: *data management, application logic,* and *preesentation.* The component that handles data management usually utilizes a DBMS for data storage, but application logic and presentation involve much more than just the DBMS itself.

1. Single Tier and Client Server Architectures:

- Data-intensive applications were combined initially into a single tier, including the DBMS, application logic, and user interface, as illustrated in Figure 7.5.

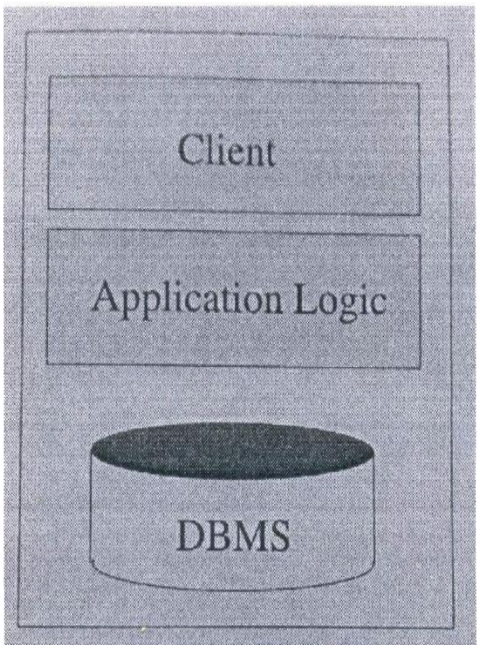


Figure 7.5: A single - Tier architecture

- 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.

Single-tier architectures drawback:

i) Users expect graphical interfaces that require much more computational power than simple dumb terminals.

ii) Centralized computation of the graphical displays of such interfaces requires much more computational power than available with a single server. Thus, single-tier architectures do not scale to thousands of users.

2.* Two-tier architectures,** 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.
- More powerful clients can implement both user interface and business logic. Clients may also implement user interface and part of the business logic, with the remaining part being implemented at

the server level. Such clients are often called **thick** clients, and this architecture is illustrated in Figure 7.7.

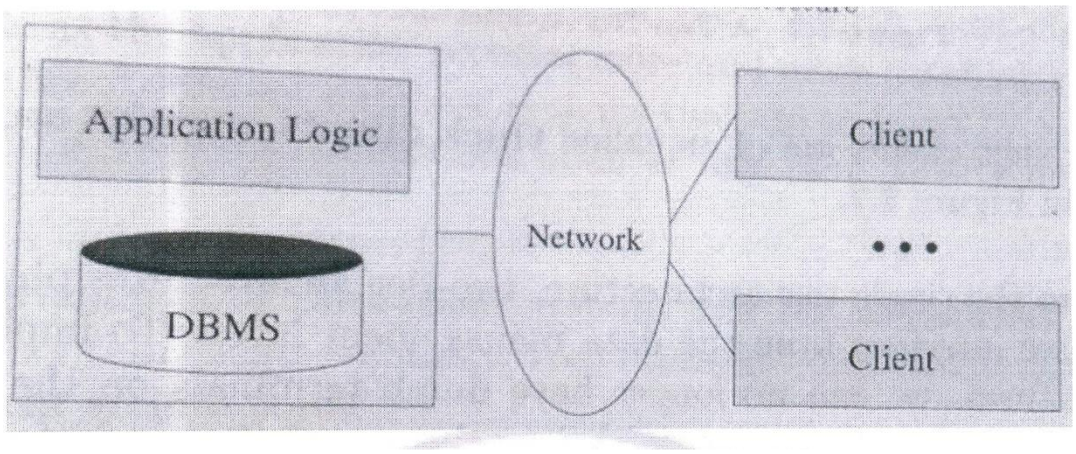


Figure 7.6: A Two server Architecture: Thin Clients

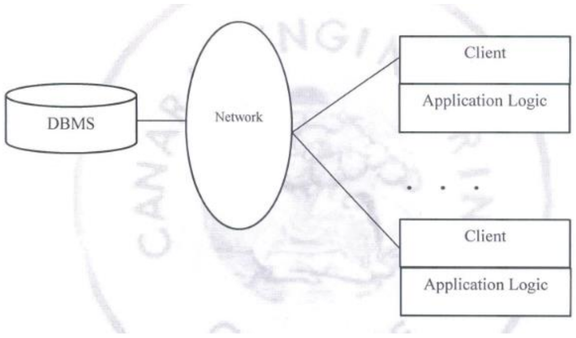


Figure 7.7: A Two tier Architecture: Thick Clients

- Two-tier architectures physically separate the user interface from the data management layer. To implement two tier architectures, computers that run sophisticated presentation code (and possibly, application logic) are required.
- Client-server development tools such as Microsoft Visual Basic and Sybase Power builder permit rapid development of client-server software, contributing to the success of the client-server model, especially the thin-client version.

Thick-client model disadvantages :

1. There is no central place to update and maintain the business logic, since the application code runs at many client sites.
2. A large amount of trust is required between the server and the clients.
3. The thick-client architecture does not scale with the number of clients. It typically cannot handle more than a few hundred clients. The application logic at the client issues SQL queries to the server and the server returns the query result to the client, where further processing takes place. Large query results might be transferred between client and server.

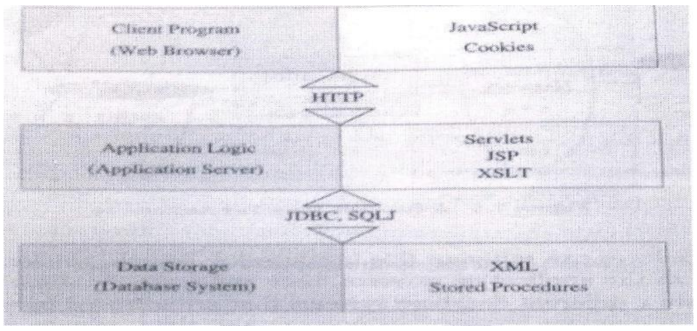
4. Thick-client systems do not scale as the application accesses more and more database systems. Assume there are *x* different database systems that are accessed by y clients, then there are x . y different connections open at any time, clearly not a scalable solution.

- ❖ The disadvantages of thick-client systems and the widespread adoption of standard, very thin clients-notably, Web browsers —have led to the widespread use thin-client architectures.

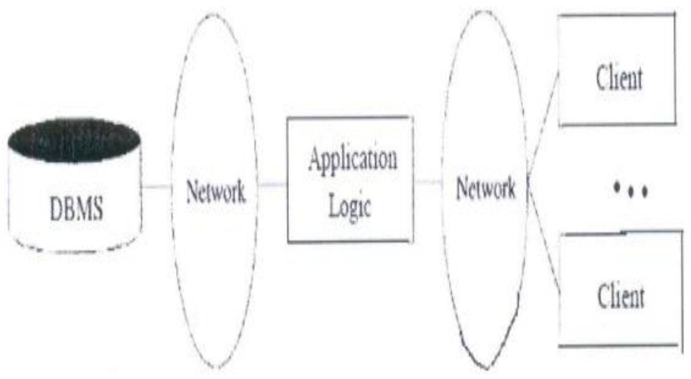
3. Three Tier Architectures:

- ❖ The thin-client two-tier architecture essentially separates presentation issues from the rest of the application.
- ❖ The three-tier architecture goes one step further, and also separates application logic from data management:
  - 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.
  - 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.
  - Data Management Tier: Data-intensive Web applications involve DBMSs.

Figure 7.9 shows the technologies relevant to each tier



Below figure shows the Three Tier Architecture:



- ❖ Different technologies have been developed to enable distribution of the three tiers of an application across multiple hardware platforms and different physical sites.

Overview of the Presentation Tier:

- ❖ At the presentation layer, we need to provide forms through which the user can issue requests, and display responses that the middle tier generates.
- ❖ The hypertext markup language (HTML) is the basic data presentation language.
- ❖ This layer of code must easily adapt to different display devices and formats. For example, regular desktops versus handheld devices versus cell phones. This adaptivity can be achieved either at the middle tier through generation of different pages for different types of client, or directly at the client through style sheets that specify how the data should be presented.
- ❖ In the latter case, the middle tier is responsible for producing the appropriate data in response to user requests, whereas the presentation layer decides *how* to display that information.

Overview of the Middle Tier:

- ❖ The middle layer runs code that implements the business logic of the application: 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.
- ❖ The middle tier code is responsible for supporting all the different roles involved in the application. For example, in an Internet shopping site implementation, we would like customers to be able to browse the catalog and make purchases, administrators to be able to inspect current inventory, and possibly data analysts to ask summary queries about purchase histories. Each of these roles can require support for several complex actions.

For example, consider that a customer who wants to buy an item (after browsing or searching the site to find it). Before a sale can happen, the customer has to go through a series of steps: She has to add items to her shopping basket, she has to provide her shipping address and credit card number (unless she has an account at the site), and she has to finally confirm the sale with tax and shipping costs added. Controlling the flow among these steps and remembering already executed steps is done at the middle tier of the application. The data carried along during this series of steps might involve database accesses, but usually it is not yet permanent (for example, a shopping basket is not stored in the database until the sale is confirmed).

3. Advantages of the Three Tier Architecture:

1. 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.
2. Thin Clients: Clients only need enough computation power for the presentation layer. Typically, clients are Web browsers.
3. 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.
4. 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. If the middle tier becomes the bottle-neck, several servers can be deployed executing the middle tier code; clients can connect to anyone of these servers, if the logic is designed appropriately.

The DBMS must be reliable for each data source to be scalable.

- Software Development Benefits: Dividing the application into parts that address presentation, data access, and business logic, gains 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.

3.12 THE PRESENTATION LAYER

Style sheets are languages that allow us to present the same webpage with different formatting for clients with different presentation capabilities. Example: Web browsers versus cell phones, or even a Netscape browser versus Microsoft's Internet Explorer.

1. HTML Forms:

- HTML forms are a common way of communicating data from the client tier to the middle tier.
- The following is the general format of a form:

```
<FORM ACTION="page.jsp" METHOD="GET" NAME="LoginForm">
</FORM>
```

- A single HTML document can contain more than one form. Inside an HTML form, we can have any HTML tags except another FORM element.

The FORM tag has three important attributes:

1. ACTION: Specifies the URI of the page to which the form contents are submitted; if the ACTION attribute is absent, then the URI of the current page is used. In the example, the form input would be submitted to the page named *page. j sp*, which should provide logic for processing the input from the form.
2. METHOD: The HTTP/1.0 method used to submit the user input from the filled-out form to the webserver. There are two choices, GET and POST.
3. NAME: This attribute gives the form a name. Naming forms is good style although its not necessary.

- Inside HTML forms, the INPUT, SELECT, and TEXTAREA tags are used to specify user input elements; a form can have many elements of each type. The simplest user input element is an INPUT field, a standalone tag with no terminating tag.

An example of an INPUT tag is the following:

```
<INPUT TYPE="text" NAME="title">
```

- The INPUT tag has several attributes. The three most important ones are:
  1. **TYPE:**The TYPE attribute determines the type of the input field. If the TYPE attribute has value text, then the field is a text input field. If the TYPE attribute has value password, then the input field is a text field where the entered characters are displayed as stars on the screen. If the TYPE attribute has value reset, it is a simple button that resets all input fields within the form to their default values. If the TYPE attribute has value submit, then it is a button that sends the values of the different input fields in the form to the server. The reset and submit input fields affect the entire form.
  2. **NAME:** The NAME attribute of the INPUT tag specifies the symbolic name for this field and is used to identify the value of this input field

when it is sent to the server. NAME has to be set for INPUT tags of all types except submit and reset. In the example, we specified title as the NAME of the input field.

3. **VALUE:** The VALUE attribute of an input tag can be used for text or password fields to specify the default contents of the field. For submit or reset buttons, VALUE determines the label of the button.

The form in Figure 7.11 shows two text fields, one regular text input field and one password field. It also contains two buttons, a reset button labeled 'Reset Values' and a submit button labeled 'Log on'. The two input fields are named, whereas the reset and submit button have no NAME attributes.

```
<FORM ACTION="page.jsp" METHOD="GET" NAME="LoginForm">
<INPUT TYPE="text" NAME="username" VALUE=" Joe"><P>
<INPUT TYPE="password" NAME="password"><P>
<INPUT TYPE="reset" VALUE="Reset Values"><P>
<INPUT TYPE="submit" VALUE="Log on"›
</FORM>
```

Figure 7.11: HTML Form with Two Text Fields and Two Buttons

Passing Arguments to Server-Side Scripts:

There are two different ways to submit HTML Form data to the webserver.

1. If the method `GET` is used, then the contents of the form are assembled into a query URL and sent to the server.

2. If the method `POST` is used, then the contents of the form are encoded as in the `GET` method, but

the contents are sent in a separate data block instead of appending them directly to the URL.

Thus, in the `GET` method the form contents are directly visible to the user as the constructed URL, whereas in the `POST` **method, the** form contents are sent inside the **HTTP** request message body and are not visible to the user.

- Using the GET method **gives users the opportunity to bookmark the page with the constructed URL**

  and thus directly jump to it in **subsequent sessions; this is not possible with the** POST **method.**
- **The choice of** GET versus POST should be determined by the application and its requirements.

❖ The encoded URI has the following form when the GET method is used:

```
action?name1=value1&name2=value2&name3=value3
```

The action is the URI specified in the ACTION attribute to the FORM tag, or the current document URI if no ACTION attribute was specified. The 'name=value' pairs are the user inputs from the INPUT fields in the form. For form INPUT fields where the user did not input anything, the name is still present with an empty value (name=).

Example: Consider the password submission form at the end of the previous section. Assume that the user inputs *'John Doe'* as username, and *'secret'* as password. Then the request URI is:

```
page.jsp?username=John+Doe&password=secret
```

❖ The user input from forms can contain general ASCII characters, such as the space character, but URIs have to be single, consecutive strings with no spaces. Therefore, special characters such as spaces, '=', and other unprintable characters are encoded in a special way.

❖ **The following three steps are performed to create a URI that has form fields:**

I. Convert all special characters in the names and values to '%xyz,' where 'xyz' is the ASCII value of the character in hexadecimal. Special characters include -----, &, %, +, and other unprintable characters. We could encode *all* characters by their ASCII value.

2. Convert all space characters to the '+' character.
3. Glue corresponding names and values from an individual HTML INPUT tag together with '=' and then paste name-value pairs from different HTML INPUT tags together using '&' to create a request URI of the form:

```
action?name1=value1&name2=value2&name3=value3
```

❖ Inorder to process the input elements from the HTML form at the middle tier, the ACTION attribute of the FORM tag needs to point to a page, script, or program that will process the values of the form fields the user entered.

2. **JavaScript:**

- JavaScript is a scripting language at the client tier with which programs can be added to webpages that run directly at the client (i.e., at the machine running the Web browser).
- JavaScript is often used for the following types of computation at the client:
    1. Browser Detection: JavaScript can be used to detect the browser type and load a browser-specific page.
    2. Form Validation: JavaScript is used to perform simple consistency checks on form fields. For example, a JavaScript program might check whether a form input that asks for an email address contains the character '@,' or if all required fields have been input by the user.
    3. Browser Control: This includes opening pages in customized windows. Examples: The pop-up advertisements at many websites are programmed using JavaScript.
- JavaScript is usually embedded into an HTML document with a special tag, the SCRIPT tag. The SCRIPT tag has the attributes:
    1. `LANGUAGE,` which indicates the language in which the script is written. For JavaScript, the language attribute is set to `JavaScript.`
    2. `SRC,` which specifies an external file with JavaScript code that is automatically embedded into the HTML document. Usually JavaScript source code files use a *.js* extension. The following fragment shows a JavaScript file included in an HTML document:

```
<SCRIPT LANGUAGE=" JavaScript" SRC="validateForm.js"> </SCRIPT>
```

The `SCRIPT` tag can be placed inside HTML comments so that the JavaScript code is not displayed verbatim in Web browsers that do not recognize the `SCRIPT` tag.

Example: A JavaScipt code that creates a pop-up box with a welcoming message.

```
<SCRIPT LANGUAGE=" JavaScript" >
<!--
    alert (" Welcome to our bookstore");
 //-->
</SCRIPT>
```

- JavaScript provides two different commenting styles:

– single-line comments that start with the '//' character, and

  – multi-line comments starting with '/*' and ending with "*/" characters.

- JavaScript has variables that can be numbers, boolean values (true or false), strings, and some other data types.
- Global variables have to be declared in advance of their usage with the keyword var, and they can be used anywhere inside the HTML documents.
- Variables local to a JavaScript function need not be declared.
- Variables do not have a fixed type, but implicitly have the type of the data to which they have been assigned.
- JavaScript has the assignment operators (=, + =, etc.), the arithmetic operators (+, *, /, %), the comparison operators (==, ! =, >=, etc.), and the boolean operators (&& for logical AND, for logical OR, and ! for negation).
- Strings can be concatenated using the '+' character.
- The type of an object determines the behavior of operators. Example: 1+1 is 2, since we are adding numbers. "1"+"1" is "11," since we are concatenating strings.

JavaScript contains the types of statements such as:

Assignments

conditional statements (if `(condition) {statements; }else {statements; })`

loops (for-loop, do-while, and `while-loop).`

JavaScript allows function creation using the function keyword:

```
function f (argl, arg2) {statements;}
```

Functions can be called from JavaScript code and functions can return values using the keyword return.
Example: A JavaScript function that tests whether the login and password fields of a HTML form are not empty.

```
<SCRIPT LANGUAGE==" JavaScript">
<!--
function testLoginEmpty()

  loginForm = document.LoginForm
  if ((loginForm.userid.value == "") II
             (loginForm.password.value == "" )) {
     alert('Please enter values for userid and password.');
     return false;
```

```
    }
    else
    return true;

  //-->
  </SCRIPT>
  <H1 ALIGN = "CENTER" >Barns and Nobble Internet Bookstore</H1>
  <H3 ALIGN = "CENTER">Please enter your userid and password:</H3>
  <FORM NAME = "LoginForm" METHOD="POST"
          ACTION= "TableOfContents.jsp"
         onSubmit="return testLoginEmpty()" >
    Userid: <INPUT TYPE="TEXT" NAME="userid"><P>
    Password: <INPUT TYPE="PASSWORD" NAME="password"><P>
    <INPUT TYPE="SUBMIT" VALUE="Login" NAME="SUBMIT">
    <INPUT TYPE="RESET" VALUE="Clear Input" NAME="RESET">
  </FORM>
```

3. Style Sheets:

- A ***style sheet*** is a method to adapt the same document contents to different presentation formats.
- A style sheet contains instructions that tell a Web browser (or whatever the client uses to display the webpage) how to translate the data of a document into a presentation that is suitable for the client's display.
  - Style sheets separate the *transformative* aspect of the page from the *rendering* aspects of the page. During transformation, the objects in the XML document are rearranged to form a different structure, to omit parts of the XML document, or to merge two different XML documents into a single document.
  - During rendering, the existing hierarchical structure of the XML document is formatted according to the user's display device.

Advantages of using style sheets:

1. We can reuse the same document many times and display it differently depending on the context.
2. We can tailor the display to the reader's preference such as font size, color style, and even level of detail.
3. We can deal with different output formats, such as different output devices (laptops versus cell Phones), different display sizes (letter versus legal paper), and different display media (paper

versus digital display).

4. We can standardize the display format within a corporation and thus apply style sheet Conventions to documents at any time. Further, changes and improvements to these display

conventions can be managed at a central place.

Two style sheet languages - XSL and CSS.

- CSS was created for HTML with the goal of separating the display characteristics of different formatting tags from the tags themselves.
- XSL is an extension of CSS to arbitrary XML documents. XSL contains a transformation language that enables us to rearrange objects besides allowing to define, ways of formatting objects.

The target files for CSS are HTML files, whereas the target files for XSL are XML files.

Cascading Style Sheets:

- A Cascading Style Sheet (CSS) defines how to display HTML elements.
- Styles are normally stored in style sheets, which are files that contain style definitions.
- Many different HTML documents, such as all documents in a website, can refer to the same CSS. Thus, the format of a website can be changed by changing a single file. This is a very convenient way of changing the layout of many webpages at the same time, and a first step toward the separation of content from presentation.

An example style sheet is shown as.

```
BODY {BACKGROUND-COLOR: yellow}
H1 {FONT-SIZE: 36pt}
H3 {COLOR: blue}
P {MARGIN-LEFT: 50px; COLOR: red}
```

It is included into an HTML file with the following line:

```
<LINK REL="style sheet" TYPE="text/css" HREF="books.css" />
```

- ❖ Each line in a CSS sheet consists of three parts - a selector, a property, and a value. They are syntactically arranged in the following way:

```
selector {property: value}.
```

- ❖ The selector is the element or tag whose format we are defining. The property indicates the tag's attribute whose value is to be set in the style sheet, and the value is the actual value of the attribute.

Consider the first line of the example style sheet shown as:

```
BODY {BACKGROUND-COLOR: yellow}
```

This line has the same effect as changing the HTML code to the following:

```
<BODY BACKGROUND-COLOR="yellow" >.
```

The value should always be quoted, as it could consist of several words. More than one property for the same selector can be separated by semicolons as shown in the example:

```
P {MARGIN-LEFT: 50px; COLOR: red}
```

XSL:

- XSL is a language for expressing style sheets.
- An XSL style sheet is, like CSS, a file that describes how to display an XML document of a given type.
- XSL shares the functionality of CSS and is compatible with it (although it uses a different syntax).
- XSL contains the **XSL Transformation** language, or XSLT, a language that allows us to transform the input XML document into a XML document with another structure.
- For example, with XSLT the order of elements that are displayed can be changed (e.g.; by sorting them), elements can be processed more than once, elements can be suppressed in one place and presented them in another, and generated text can be added to the presentation.
- XSL also contains the **XML Path Language (XPath),** a language that allows us to refer to parts of an XML document. XSL also contains XSL Formatting Object, a way of formatting the output of an XSL transformation.

3.13 THE MIDDLE TIER

The first generation of middle-tier applications was stand-alone programs written in a general-purpose programming language such as C, C++, and Perl. The overheads include starting the application every time it is invoked and switching processes between the webserver and the application. Therefore, such interactions do not scale to large numbers of concurrent users. This led to the development of **the application server,** which provides the run-time environment for several technologies that can be used to program middle-tier application components.

The Common Gateway Interface is a protocol that is used to transmit arguments from HTML forms to application programs running at the middle tier. The technologies for writing application logic at the **middle** tier are Java servlets and Java Server Pages. Another important functionality is the maintenance of state in the middle tier component of the application as the client component goes through a series of steps to complete a transaction (for example, the purchase of a market basket of items or the reservation of a flight).

1. CGI: The Common Gateway Interface:

- The Common Gateway Interface connects HTML forms with application programs.
- It is a protocol that defines how arguments from forms are passed to programs at the server side. Programs that communicate with the webserver via CGI are often called **CGI scripts,** since many such application programs were written in a scripting language such like Peri.

Example: A program that interfaces with an HTML form via CGI.

The sample webpage shown in Figure 7.14 contains a form where a user can fill in the name of an author. If the user presses the 'Send it' button, the Perl script *'findBooks.cgi'* shown in Figure 7.14 is executed as a separate process.

```
<HTML><HEAD><TITLE>The Database Bookstore</TITLE></HEAD>
<BODY>
<FORM ACTION="find books.cgi" METHOD=POST>
   Type an author name:
   <INPUT TYPE="text" NAME="authorName"
          SIZE=30 MAXLENGTH=50>
   <INPUT TYPE="submit" value="Send it">
   <INPUT TYPE="reset" VALUE="Clear form" >
</FORM>
</BODY></HTML>
```

Figure 7.14: A Sample Web Page Where Form Input Is Sent to a CGI Script

The CGI protocol defines how the communication between the form and the script is performed. Figure 7.15 illustrates the processes created when using the CGI protocol.

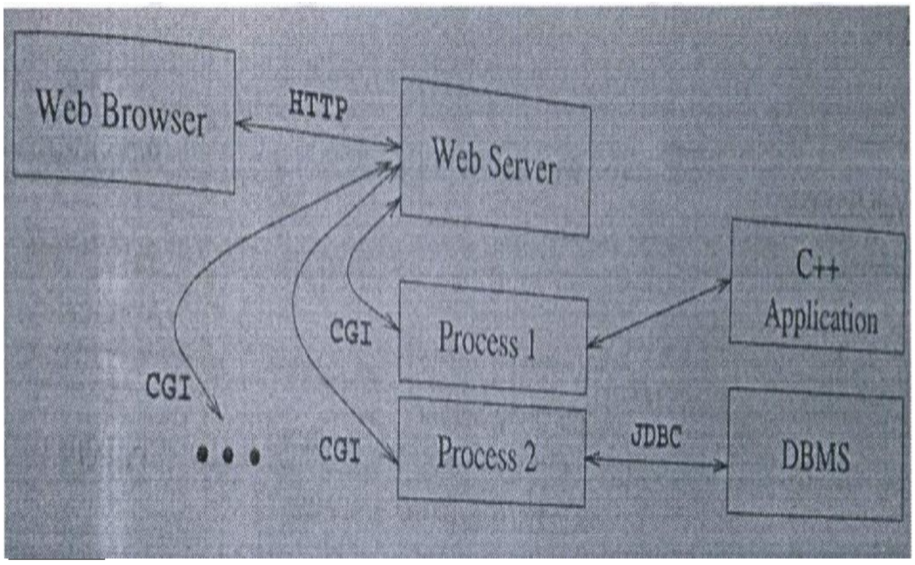


Figure 7.15: Process Structure with CGI scripts

```
#!/usr/bin/perl
use CGI;

### part 1
$dataIn  =  new  CGI;  $dataIn->header();
$authorName = $dataIn->param('authorName');

### part 2
print ("<HTML><TITLE>Argument passing test</TITLE> ") ;
print ("The user passed the following argument: ") ;
print ("authorName: ", $authorName);
### part 3
print ("</HTML>");
exit;
```

Figure 7.16: A Simple Perl Script

Perl is an interpreted language that is often used for CGI scripting and many Perl libraries, called **modules,** provide high-level interfaces to the CGI protocol. One such library, called the **DBI library** is used in the example. The CGI module is a convenient collection of functions for creating CGI scripts. In part 1 of the sample script, the argument of the HTML form is extracted and that is passed along from the client as follows:

```
$authorName = $datain->param(`authorName ) ;
```

The parameter name authorName was used in the form in Figure 7.14 to name the first input field. Conveniently, the CGI protocol abstracts the actual implementation of how the webpage is returned to the Web browser; the webpage consists simply of the output of our program, and we start assembling the output HTML page in part 2. Everything the script writes in print statements is part of the dynamically constructed webpage returned to the Browser. In part 3, the closing format tags are appended to the resulting page.

2. Application Servers:

- Application logic can be enforced through server-side programs that are invoked using the CGI protocol. Since each page request results in the creation of a new process, this solution does not scale well to a large number or simultaneous requests. This performance problem led to the development of specialized programs called application servers.
- An application server maintains a pool of threads or processes and uses these to execute requests. Thus, it avoids the startup cost of creating a new process for each request.
- Application servers have evolved into flexible middle-tier packages that provide many functions in addition to eliminating the process-creation overhead. They facilitate concurrent access to several heterogeneous data sources (e.g., by providing **JDBC** drivers), and provide **session management services.**

- ❖ **Users** expect the system to maintain continuity during such a multistep session. Several session identifiers such as cookies, URI extensions, and hidden fields in HTML forms can be used to identify a session. Application s:2:¯vers provide functionality to detect when a session starts and ends and keep track of the sessions of individual users. They also help to ensure database access by supporting a general user-id mechanism.

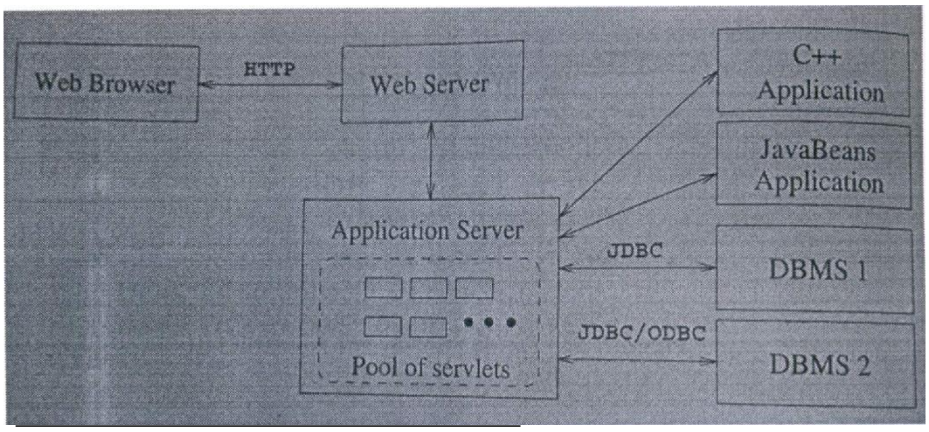


Figure 7.17: Process structure in the application server Architecture

- ❖ The execution of business logic at the webserver's site, **serverside processing,** has become a standard model for implementing more complicated business processes on the Internet.

3. Servlets:

- ❖ **Java servlets** are pieces of Java code that run on the middle tier, in either webservers or application servers.

- Servlets are truly platform-independent.
- ❖ Since servlets are Java programs, they are very versatile. For example, servlets can build webpages, access databases, and maintain state. Servlets have access to all Java APIs, including JDBC.
- ❖ All servlets must implement the Servlet interface.
- ❖ In most cases, servlets extend the specific HttpServlet class for servers that communicate with clients via HTTP.
- ❖ The HttpServlet class provides methods such as doGet and doPost to receive arguments from HTML forms, and it sends its output back to the client via HTTP. Servlets that communicate through other protocols (such as ftp) need to extend the class GenericServlet.
- ❖ Servlets are compiled Java classes executed and maintained by a servlet container.
- ❖ The servlet container manages the lifespan of individual servlets by creating and destroying them.

- Servlets can respond to any type of request but they are commonly used to extend the applications hosted by webservers. For such applications, there is a useful library of HTTP-specific servlet classes.
- Servlets usually handle requests from HTML forms and maintain state between the client and the server.

A template of a generic servlet structure is shown below:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletTemplate extends HttpServlet f
public void doCet(HttpServletRequest request,
                HttpServletResponse response)throws
                ServletException, IOException fPrintriter cut =
                response.getWriter();
         // Use 'out' to send content to browser
        out.println("Hello World");
```

data depending on the HTTP transfer method. The service ( ) method is not overridden unless we want to program a servlet that handles both HTTP POST and HTTP GET requests identically.

The example, shown in Figure 7.19 illustrates how to pass arguments from an HTML form to a servlet.

```
import java.io. *;
import javax.servlet. *;
import javax.servlet.http. *;
import java.util.*;
public class ReadUserName extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
response.setContentType('j textjhtml'j);
PrintWriter out = response.getWriter();
out.println("<BODY>\n" +
"<Hi ALIGN=CENTER> Username: </Hi>\n" +
"<UL>\n" +
" <LI>title: "
+ request.getParameter("userid") + "\n" +
+ request.getParameter("password'j) + "\n
j
' +
1</UL>\n" +
1</BODY></HTML>")j
}
public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
doGet(request, response);
}
}
```

4. JavaServer Pages:

❖ JavaServer pages (JSPs) interchange the roles of output and application logic.

- ❖ JavaServer pages are written in HTML with servlet-like code embedded in special HTML tags. Thus, in comparison to servlets, JavaServer pages are better suited to quickly building interfaces that have some logic inside, whereas servlets are better suited for complex application logic.
- ❖ The middle tier handles JavaServer pages in a very simple way: They are usually compiled into a servlet, which is then handled by a servlet container analogous to other servlets.

```
<html>
   <head>
      <title>Using GET Method to Read Form Data</title>
   </head>
   <body>
      <h1>Using GET Method to Read Form Data</h1>
      <ul>
         <li><p><b>First Name:</b>
            <%= request.getParameter("first_name")%>
            </p></li>
         <li><p><b>Last Name:</b>
            <%= request.getParameter("last_name")%>
            </p></li>
      </ul>
   </body>
</html>
```

Reading Form Parameters in JSP

5. Maintaining State:

There are basically two choices where we should maintain state since HTTP protocol cannot maintain state. We can maintain state in the middle tier, by storing information in the local main memory of the application logic, or even in a database system. Alternatively, we can maintain state on the client side by storing data in the form of a *cookie.*

Maintaining State at the Middle Tier:

- ❖ At the middle tier, there are several choices as to *where* we maintain state.
  - ➢ The state could be stored at the bottom tier, in the database server. The state survives crashes of the system, but a database access is required to query or update the state, a potential performance bottleneck.
  - ➢ An alternative is to store state in main memory at the middle tier. The drawbacks are that this information is volatile and that it might take up a lot of main memory.
  - ➢ The state can also be stored in local files at the middle tier, as a compromise between the first two approaches.
- ❖ A rule of thumb is to use state maintenance at the middle tier or database tier only for data that needs to persist over many different user sessions. Examples of such data are past customer orders, click-stream data recording a user's movement through the website, or other permanent choices that a user makes, such as decisions about personalized site layout, types of messages the user is willing to receive, and so on. As these examples illustrate, state information is often centered around users who interact with the website.

Maintaining State at the Presentation Tier: Cookies:

- ❖ The state can be stored at the presentation tier and passed to the middle tier with every HTTP request. We essentially work around the statelessness of the HTTP protocol by sending additional information with every request. Such information is called a cookie.
- ❖ A **cookie** is a collection of <*name, value*> - pairs that can be manipulated at the presentation and middle tiers.
- ❖ Cookies are easy to use in Java servlets and JavaServer Pages and provide a simple way to make non-essential data persistent at the client. They survive several client sessions because they persist in the browser cache even after the browser is closed.
- ❖ Disadvantage of cookies:
  - ■ Cookies are often perceived as being invasive, and many users disable cookies in their Web browser. Browsers allow users to prevent cookies from being saved on their machines.
  - ■ The data in a cookie is currently limited to 4KB, but for most applications this is not a bad limit. needs to have sufficient consistency checks to ensure that the data in the cookie is valid.
- ❖ Cookies can be used to store information such as the user's shopping basket, login information, and other non-permanent choices made in the current session.

The Servlet Cookie API:

- A cookie is stored in a small text file at the client and. contains <*name, value*> pairs, where both name and value are strings. We create a new cookie through the Java Cookie class in the middle tier application code:

```
Cookie cookie = new Cookie("username" ,"guest");
cookie.setDomain("www.bookstore.com");
cookie.setSecure(false);                        / /no SSL required
cookie. setMaxAge (60*60*24*7*31) ;            / /one month lifetime
response.addCookie(cookie);
```

First, a new Cookie object is created with the specified <*name, value*> pair. Then the attributes of the cookie are set. The cookie is added to the request object within the Java servlet to be sent to the client. Once a cookie is received from a site (*www.bookstore.com* in this example), the client's Web browser appends it to all HTTP requests it sends to this site, until the cookie expires. We can access the contents of a cookie in the middle-tier code through the request object `getCookies ( )` method, which returns an array of Cookie objects.

+ Some of the most common attributes of cookies are:

  - ■ set `Domain` and `getDomain:` The domain specifies the website that will receive the cookie. The default value for this attribute is the domain that created the cookie.

- setSe cure and getSecure: If this flag is true, then the cookie is sent only if we are using a secure version of the HTTP protocol, such as SSL.
- setMaxAge and getMaxAge: The MaxAge attribute determines the lifetime of the cookie in seconds. If the value of MaxAge is less than or equal to zero, the cookie is deleted when the browser is closed.
- setName and getName: These functions allow us to name the cookie.
- setValue and getValue: These functions allow us to set and read the value of the cookie.

The following code fragment reads the array and looks for the cookie with name 'use rname.'

```
Cookie[] cookies = request.getCookies();
String theUser;
for(int i=0; i < cookies.length; i++) {
     Cookie cookie = cookies[i];
      if (cookie.getName().equals("username"))
          theUser = cookie.getValue();
```

A simple test can be used to check whether the user has turned off cookies: Send a cookie to the user, and then check whether the request object that is returned still contains the cookie. A cookie should never contain an unencrypted password or other private, unencrypted data, as the user can inspect, modify, and erase any cookie at any time, including in the middle of a session. The application logic